НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

ФАКУЛЬТЕТ ІНФОРМАТИКИ ТА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

Кафедра автоматизованих систем обробки інформації і управління

«На правах рукопису»

УДК _____

«До захисту допущено»
В.о. завідувача кафедри
_____ *О.А.Павлов*
(підпис) (ініціали, прізвище)
“___”_____2019 р.

Магістерська дисертація

зі спеціальності *121 «Інженерія програмного забезпечення»*

на тему: «*Архітектура програмного забезпечення для платформи iOS*»

Виконала: студентка VI курсу, групи *ІП-82мп*

Салій Ольга Олексіївна (прізвище, ім’я, по батькові) _____ (підпис)

Науковий керівник *доц., к.т.н. Ліщук К.І.* (посада, науковий ступінь, , вчене звання, прізвище та ініціали) _____ (підпис)

Консультант *доц., к.т.н., Ліщук К.І.* (посада, науковий ступінь, вчене звання, прізвище та ініціали) _____ (підпис)

Рецензент *доц. каф. ТК, к.т.н., доц. Ткач М.М.* (посада, науковий ступінь, вчене звання, прізвище та ініціали) _____ (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.

Студент _____
(підпис)

Київ – 2019 року

РЕФЕРАТ

Актуальність теми: У багатьох випадках мобільні застосунки розробляються для вирішення бізнесових задач, і в свою чергу для бізнесу важливі часові дедлайни, швидкість та якість. Зазвичай ці мобільні за стосунки є частиною клієнт-серверної архітектури, де клієнтом виступає як раз мобільний за стосунок. Такий тип за стосунків є найпопулярнішим в наш час серед українських iOS розробників, і в тому саме для цього типу є сенс і потреба у створенні дійсно ефективного й простого архітектурного підходу. Основною проблемою є те, що всі архітектурні підходи описані дуже поверхнево, і це призводить до того, що кожний розробник інтерпретує визначення архітектур по своєму і використовує свої підходи. Це, в свою чергу приводить до створення багатьох різних реалізацій, які частково або зовсім не відповідають початковій задумці.

мета дослідження. Основна мета даної роботи полягає в дослідженні можливих архітектурних рішень для мобільної платформи iOS та розробці продуманої, гнучкої, і, в той же час, простої для розуміння архітектури програмного забезпечення.

Для реалізації поставленої мети були сформульовані **наступні завдання**:

- аналіз існуючих архітектурних підходів для платформи iOS;
- вибір необхідних програмних засобів для розробки програмного забезпечення;
- проектування та розробка архітектурного підходу для розробки мобільних застосунків на платформі iOS;
- розробити мобільний за стосунок, використовуючи запропонований архітектурний підхід;
- дослідження ефективності запропонованого архітектурного рішення;

Об’єкт дослідження - процес розробки мобільних застосунків на платформі iOS.

Предмет дослідження – архітектурні підходи для розробки мобільних застосунків на платформі iOS.

Наукова новизна.

Найбільш суттєвими науковими результатами магістерської дисертації є:

- запропоновано вдосконалене архітектурне рішення для розробки мобільних застосунків на платформі iOS.
- запропоновано підхід «Конструктор» до створення iOS застосунків.

Практичне значення отриманих результатів визначається тим, що запропонований архітектурний підхід доведено до практичної реалізації. Розроблено програмний продукт, який демонструє основні переваги запропонованого архітектурного рішення. модифікований алгоритм ефективно виконує операцію поєднання розподілених даних.

Зв’язок роботи з науковими програмами, планами, темами. Робота виконана в рамках теми «Методи та технології в задачах пошуку та збереження даних. Державний реєстраційний номер 0117U000915».

Апробація: Основні положення роботи доповідались і обговорювались на II всеукраїнській науково-практичній конференції молодих вчених та студентів «Інформаційні системи та технології управління» (ІСТУ-2019)

Ключові слова: АРХІТЕКТУРА, АРХІТЕКТУРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, МОБІЛЬНИЙ ЗАСТОСУНОК, iOS, MVC,MVP, MVVM.

ABSTRACT

Actuality of theme: In many cases mobile applications are designed to meet business challenges and time, speed, and quality are important to businesses. Typically, these mobile applications are part of a client-server architecture, where the client is just a mobile application. This type of application is the most popular in our time among Ukrainian iOS developers and that is why this type requires the creation of a really effective and simple architectural approach. The main problem is that all architectural approaches are described very superficially which leads to the fact that each developer interprets architectures in their own way and uses their own approaches. This, in turn, leads to the creation of many different implementations that are partially or completely inconsistent with the original idea.

The aim of the research: The main purpose of this work is to explore possible architectural solutions for the iOS mobile platform and to develop thoughtful, flexible and, at the same time, easy to understand software architectures

To achieve this goal, the **following tasks** were formed:

- analysis of existing architectural approaches for the iOS platform;
- selection of necessary software for software development;
- designing and developing an architectural approach for developing mobile applicationsons on iOS platforms;
- develop a mobile application using the proposed architectural approach;
- study of the effectiveness of the proposed architectural solution.

The object of the research is the process of developing mobile applications on the iOS platform.

The subject of the research is architectural approaches for developing mobile applications for the iOS platform.

Scientific Novelty: The most significant scientific results of the master's thesis are:

- an advanced architectural solution for developing mobile applications on the iOS platform;
- “Designer” approach for creating iOS applications is suggested.

The practical value of the results is determined by the fact that the proposed architectural approach is brought to practical implementation. A software product has been developed that demonstrates the main advantages of the proposed architectural solution.

Relationship with working with scientific programs, plans, topics: The work was performed within the theme “Methods and technologies in the tasks of data retrieval and storage. State Registration Number 0117U000915”.

Keywords: ARCHITECTURE, DESIGN, DESIGN APPROACH, IOS, MVC, MVP, MVVM, MOBILE APPLICATOIN.

ЗМІСТ

ВСТУП

З розвитком сучасних технологій, які дозволяють створювати різні мобільні пристрої, ринок програмних продуктів отримує потужний стимул до розширення і розвитку. Сучасну мобільну техніку люди носять з собою завжди і скрізь. Смартфони, планшети та інші пристрої мають значиму роль як в повсякденному житті, так і на роботі: стало можливим легко і швидко прочитати файл, зайти на пошту, надрукувати документ за допомогою мережевого принтера і багато іншого. У зв'язку зі збільшенням продажів мобільних пристроїв, поступово сформувалося окремий напрямок програмного забезпечення - мобільні додатки. Технології вливаються в бізнес-процеси сучасних підприємств все з більшою силою, поступово стаючи все більш зручними у використанні і захоплюють все більше галузей в компанії. Але постає питання про оперативність і зручність отримання даних - і тут на допомогу електронного бізнесу приходять мобільні технології. Важко назвати бізнес, який би не було порушено наростаючою хвилею пристроїв, додатків і технологій, все більш активно використовуваних покупцями, партнерами і співробітниками.

Мобільні технології - найостанніша і найбільш актуальна тенденція розвитку ринку інформаційних технологій для автоматизації бізнес-процесів. Смартфони і планшети все більше проникають в корпоративну середу і перетворюються для багатьох керівників і співробітників з особистого мобільного пристрою в робочий інструмент. Багато компаній і організації замислюються про те, щоб централізовано впроваджувати і використовувати ці технології в інтересах бізнесу.

Трафік мобільного інтернету зростає з кожним роком. Мобільні додатки мають перевагу над комп'ютерними, тим, що ними можна користуватися в будь-якому куточку світу: по дорозі на роботу, в кафе на обіді, на прогулянці в парку. Мобільне програмне забезпечення вирішує більшість повсякденних завдань людини, що дозволяє звести час витрачений на рутинні справи до

нулю. Тому розробка мобільних застосунків стає дедалі затребованою та серйозною.

Найголовнішим етапом розробки програмного забезпечення є вибір архітуктури. Бо саме це стає базою для подальшого росту і розвитку програмного забезпечення. Погано продумана ахрітуктура може привести до поганої розширюваності, до неможливості підтримки застосунку, а й у крайньому випадку призведе до неминучої необхідносту переписати все з нуля. Дуже часто, приступаючи до розробки програми, розробники під тиском менеджера беруть один з поширених архітектурних підходів і швидко розробляють, щоб отримати живий прототип за день-другий. Начебто все працює, випускається в реліз і всі задоволені. Ось тільки потім, коли постає питання про підтримку, рефакторінгу і введенні нових особливостей та функціональностей, виявляється, що у застосунку тонна коду і в написаний код впровадження нових функціональністей вияляється важче, ніж переписати все з нуля.

Тому метою цієї дисертації є створення продуманої, гнучкої і, в той же час, простої для розуміння архітектури програмного забезпечння для мобільної платформи iOS.

1 ТЕОРЕТИЧНІ ПОЛОЖЕННЯ

1.1 Архітектура програмного забезпечення

Архітектура програмного забезпечення (англ. Software architecture) - це структура програми або обчислювальної системи, яка включає програмні компоненти, видимі зовні властивості цих компонентів, а також відносини між ними.[1]

Процес проектування архітектури програмного забезпечення включає в себе збір необхідної підтримки, їх аналіз та створення проекту для компонента програмного забезпечення у відповідність до вимог. Успішна розробка ПО повинна забезпечувати баланс неминучих компромісів внаслідок суперечать вимог; відповідати принципам проектування і рекомендованим методам, виробленим з часом; і доповнювати сучасне обладнання, мережі і системи управління. Надійна архітектура програмного забезпечення вимагає значного досвіду в теоретичних і практичних питаннях, а також уяви, необхідного для перетворення бізнес-сценаріїв і вимог, які можуть здаватися незрозумілими, в надійні і практичні робочі проекти.

Архітектура програмного забезпечення включає в себе визначення структурованого рішення, що відповідає всім технічним і робочим вимогам, одночасно оптимізуючи загальні атрибути якості, такі як продуктивність, безпеку і керованість. Сюди входить серія рішень, заснованих на широкому діапазоні факторів, і кожне з цих рішень може значно впливати на якість, продуктивність, підтримувані і загальний успіх програмного забезпечення.

Сучасне програмне забезпечення рідко буває автономним. Як мінімум, в більшості випадків воно буде взаємодіяти з джерелом даних, наприклад корпоративною базою даних, що надає інформацію, з якої працюють користувачі програмного забезпечення. Зазвичай сучасне програмне забезпечення також має своєї компетенції повідомляти керівництву і

мережевими функція, щоб здійснювати, отримання та публікації інформації і надання інтегрованих середовищ роботи користувачів. Без відповідної архітектури може бути складно, якщо взагалі можливо, здійснити розгортання, експлуатацію, обслуговування і успішну інтеграцію з іншими системами; крім того, вимоги користувачів не будуть дотримані.

Архітектуру програмного забезпечення можна розглядати як зіставлення між метою компонента ПО і відомостями про реалізацію в коді. Правильне розуміння архітектури забезпечить оптимальний баланс вимог і результатів. Програмне забезпечення з добре продуманою архітектурою буде виконувати зазначені завдання з параметрами вихідних вимог, одночасно забезпечуючи максимально високу продуктивність, безпека, надійність і багато інших чинників.

На найвищому рівні проект архітектури повинен надавати структуру системи, але приховувати деталі реалізації; охоплювати всі випадки застосування і сценарії; намагатися враховувати вимоги всіх зацікавлених осіб; і задовольняти настільки, наскільки можливо, всім функціональні вимогам і вимогам до якості.

Вимоги до сучасного програмного забезпечення стають все більш складними, оскільки користувачі очікують все більше від додатків. Можливостей простих автономних настільних додатків більше не достатньо для більшості комерційних і ділових ситуацій. У світі розвиненою зв'язку додатки повинні взаємодіяти з іншими додатками і службами, а також працювати в різних середовищах, наприклад в хмарі, і на портативних пристроях. На зміну поширеним в минулому монолітних архітектур прийшло компонентне сервіс-орієнтоване програмне забезпечення, що використовує платформи, операційні системи, хост-додатки і мережі для реалізації функцій, про які було відомо всього кілька років тому.

Ці труднощі впливають не тільки на архітектуру, але також і на розгортання, обслуговування та адміністрування програмного забезпечення. Сукупна вартість володіння (ТСО) програмного забезпечення тепер в

основному складається з витрат, що виникають після розгортання. Додаток з добре продуманою архітектурою забезпечить мінімальну сукупну вартість володіння завдяки зниженню витрат і часу, необхідних на розгортання додатки, забезпечення його роботи, оновлення для задоволення мінливих вимог і усунення проблем. Адміністрування та підтримка користувачів також будуть спрощені.

Успішне програмне забезпечення також має відповідати кільком важливим критеріям. Воно повинно забезпечувати безпеку, щоб додаток і його дані були захищені від атак зловмисників і випадкових помилок. Воно повинно бути стійким і надійним для мінімізації збоїв і відповідних витрат. Воно повинно працювати з необхідними параметрами у відповідність до вимог користувачів, такими як максимальний час відгуку або певна робоче навантаження. Воно повинно бути простим в підтримці для зниження витрат на адміністрування та підтримку і досить розширюваним для включення неминучих змін і оновлень, які з часом будуть потрібні.

З усіма цими фактори пов'язані деякі компроміси. Наприклад, реалізація найбільш безпечних механізмів з використанням складного шифрування вплине на продуктивність. Реалізація безлічі параметрів конфігурації та оновлення може ускладнити розгортання і адміністрування. Крім того, чим складніше архітектура, тим дорожче її реалізація. Правильна архітектура повинна забезпечувати баланс цих факторів з метою отримання оптимального результату для певного сценарію.

Архітектура програмного забезпечення починається з набору вимог. Вони можуть бути виражені у формі діаграм, блок-схем процесу, моделей або документованих списків завдань експлуатації, які має виконувати програмне забезпечення. Зазвичай клієнт або партнер також висловлює менш точні вимоги, такі як зовнішній вигляд або спосіб роботи певних користувальницьких інтерфейсів для часто зустрічаються завдань. Вимоги також повинні включати в себе інформацію про існуючий програмному забезпеченні, системах, обладнанні і мережах, з якими буде взаємодіяти нове програмне забезпечення;

та інші фактори, такі як план розгортання і обслуговування, і, звичайно ж, доступний бюджет проекту.

Розробник архітектури програмного забезпечення повинен враховувати потреби клієнта. Однак загальний термін «клієнт» зазвичай складається з трьох суперечать областей

відповідальності: бізнес-вимоги, вимоги користувача і системні вимоги. Бізнес-вимоги зазвичай визначають діапазон таких факторів, як бізнес-процеси, фактори продуктивності (такі як безпека, надійність і пропускну здатність), а також бюджетні обмеження і обмеження на витрати. Вимоги користувача включають в себе дизайн інтерфейсу, виробничі можливості і простоту використання програмного забезпечення. Системні вимоги мають на увазі можливості і обмеження обладнання, мережі і середовища виконання.

Кожен архітектор програмного забезпечення має власний підхід до збору та аналізу вимог, а також визначенню архітектури. Однак їм часто доводиться відповідати на такі питання: «Як користувачі будуть працювати з додатком?»; «Як додаток буде розгорнуто у виробничому середовищі і управлятися?»; «Які вимоги атрибутів якості для додатка, такі як безпека, продуктивність, паралелізм, інтернаціоналізація та конфігурація?»; «Яка повинна бути архітектура додатки для забезпечення гнучкості та зручності в обслуговуванні з плином часу?» І «Які архітектурні тенденції можуть впливати на додаток зараз і після його розгортання?».

Останнє питання одночасно важливе і цікаве. Гарна архітектура програмного забезпечення не тільки відповідає поточним вимогам клієнтів, але і буде відповідати таким вимогам в доступному для огляду майбутньому. Це впливає на рішення, що приймаються розробником архітектури щодо обладнання, компонентів, платформ, середовищ виконання, програмних систем управління і безлічі інших функцій, вбудованих в програмне забезпечення або з якими воно має інтегруватися.

Як і більшість завдань в світі проектування і розробки програмного забезпечення, розробка архітектури - це одночасно попереджувальний і

ітеративний процес. Багато початкові завдання, такі як аналіз вимог, технічне дослідження і визначення цілей, зазвичай виконуються на початку процесу. Наступний етап - визначення ключових сценаріїв для архітектури. Це основні вимоги, яким має відповідати програмне забезпечення, і обмеження, в рамках яких воно повинно працювати. На підставі цієї інформації розробник архітектури може створити огляд додатки. Цей огляд охоплює такі високорівневі відомості, як тип програми (для Інтернету, телефонів, настільний або хмарне), архітектура розгортання (зазвичай багаторівнева архітектура, компоненти якої зв'язуються через кордони обладнання та мережі), відповідні застосовні стилі архітектури (наприклад, n-рівнева, клієнт -сервер або сервіс-орієнтована) і технології реалізації, найкраще підходять для сценарію.

Після цього розробник може приступити до створення кандидатів архітектури, які відповідають високорівневим і найбільш важливим вимогам, визначеним раніше. Після цього проект архітектури перевіряється і тестується в ключових сценаріях, часто в поєднанні з відгуками клієнтів і пробними або тестовими версіями, для забезпечення отримання оптимального результату. Це малоймовірно при першій ітерації, але при повторенні циклу буде досягнуто відповідність проекту вимогам і ключовим сценаріями.

У міру деталізації проекту і визначення окремих завдань і компонентів архітектор може уточнити і додати подробиці на кожному етапі. Наприклад, після визначення архітектурного стилю і підходу до розгортання архітектор може прийняти рішення про зв'язок рівнів і компонентів. Сюди може входити вибір протоколу на основі існуючих і майбутніх вимог, а також прийняття до уваги оглядів нових технологій і можливостей, визначених у майбутніх стандартах.

Остаточний продукт роботи архітектора - це зазвичай набір схем, моделей і документів, що визначають додаток з декількох точок зору, щоб при об'єднанні вони могли надати розробникам, групам тестування, адміністраторам і управління всю необхідну інформацію для реалізації проекту. Ця інформація буде описувати структуру і розміщення компонентів і рівнів

додатки; спосіб обробки горизонтального перетину ієрархії, наприклад, протоколювання і перевірки; плани тестування і розгортання; і документацію для розробників, адміністраторів і персоналу служби підтримки.

В остаточному проекті також можуть бути вказані атрибути якості, яким має відповідати додаток. Це результат прийнятих рішень і компромісів розробника архітектури по консультації з клієнтом. До них відносяться визначення вимог безпеки і план реалізації безпеки, необхідна масштабованість і продуктивність при розгортанні на цільовій платформі, способи реалізації можливості обслуговування і розширюваності і функції забезпечення взаємодії з іншими системами.

1.2 Загальні положення про платформу iOS

Мобільнии застосунок - програмне забезпечення, призначене для роботи на смартфонах, планшетах та інших мобільних пристроях. Багато мобільних додатків встановлені на самому пристрої або можуть бути завантажені на нього з онлаин магазинів мобільних додатків, таких як App Store, Google Play, Windows Phone Store та інших, безкоштовно або за плату. [2]

Існує багато різних мобільних операциних систем, наипопулярніші серед них: Symbian, Android, iOS, Blackberry OS, Windows. Надалі будемо розглядати тільки платформу iOS.

Платформа iOS - це власницька мобільна операциона система від Apple. Розроблена спочатку для iPhone, вона стала операциною системою також для iPod Touch, iPad і Apple TV. Apple не дозволяє роботу ОС на мобільних телефонах інших фірм. Користувацькии інтерфеис iOS заснованиі на концепції прямої маніпуляції з використанням жестів Multi-Touch. Елементи інтерфеису управління складаються з повзунків, перемикачів і кнопок. Він призначении для безпосереднього контакту користувача з екраном пристрою. Внутрішніи акселерометр використовуються деякими програмами для реагування на струшування пристрою, яке є також загальною

командою скасування, або обертання пристрою у трьох вимірах, що є загальною командою перемикання між книжковим та альбомним режимами.

Як операціи□на система iOS була представлена з iPhone на Macworld Conference & Expo 9 січня 2007 року і випущена в червні того ж року. Спершу, Apple не вказувала і□і□ ім'я, просто заявивши, що "iPhone використовує OS X". Спочатку, сторонні програми не підтримувалися. Стів Джобс заявив, що розробники можуть створювати веб-програми, що „будуть вести себе, як рідні програми на iPhone“. 17 жовтня 2007 року Apple оголосила, що ріднии□ SDK знаходиться в стадіі□ розробки, і що вони планують поставити и□ого „в руки розробників у лютому“. 6 березня 2008 року Apple випустила першу бета-версію, а також нове ім'я для операціи□ноі□ системи: iPhone OS. Продажі мобільних пристроі□в Apple викликали інтерес до SDK. Apple також продала більше одного мільи□она iPhones під час курортного сезону 2007. У червні 2010 року, Apple переи□менувала iPhone OS на iOS. Назва iOS користувалася компанією Cisco вже більше десяти років на маршрутизаторах Cisco. Для того, щоб уникнути будь- якого потенціи□ного позову, Apple ліцензувала торгову марку iOS у Cisco. [3]

1.3 Розробка для платформи iOS

На даний момент існує найбільш популярні операційної системи (ОС) для мобільних пристроїв: ОС iOS - компанії Apple і ОС Android - компанії Google. У четвертому кварталі 2016 року спільна частка мобільних ОС від Apple і Google на світовому ринку становить 98,4%, що на 2% більше, ніж аналогічний показник в минулому році - 96,4%.

Надалі, як можна припустити, частка Android і iOS на ринку буде лише збільшуватися. Платформа Windows Phone, яку ніяк не можна назвати конкурентом мобільним ОС від Apple і Google, продовжує здавати позиції, а інші все так же топчуться на місці.

Візьмемо за основу мобільну ОС iOS і розглянемо принципи розробки програмних продуктів для мобільних пристроїв на її прикладі. Для грамотного створення програми необхідно враховувати принципи простоти, зручності і інтуїтивно-зрозумілого управління. При проектуванні інтерфейсу необхідно розробити сценарії того, як користувач буде використовувати програму, наприклад які жести найбільш зручні. ОС iOS підтримує безліч способів маніпуляції з сенсорним екраном завдяки функції Multi-touch, що підвищує зручність роботи з програмним продуктом.

Написання програми під ОС iOS здійснюється на комп'ютерах Mac компанії Apple в робочому середовищі Xcode, тут є можливість написання програми на двох мовах: Objective-C або Swift.

1.4 Середовище для розробки на iOS

Зупинимося докладніше на середовищі розробки Xcode - це додаток для Mac, призначене для розробки інших додатків для Mac і iOS. Його можна завантажити безкоштовно з App Store для Mac.

Xcode тісно інтегрований з framework Cocoa, який складається з бібліотек, API, і середовищ, які формують шар розробки для всіх Mac OS X. Даний набір інструментів включає в себе:

- Xcode IDE (для кодування, створення і налагодження додатків);
- Interface Builder (для розробки користувальницького інтерфейсу);
- інструменти для аналізу поведінки і продуктивності;
- десятки додаткових інструментів. [4]

1.5 Мови програмування для розробки на iOS

Розглянемо докладніше мови програмування. Почнемо з Objective-C, який є розширенням мови програмування C і надає об'єктно-орієнтовані можливості, а так само динамічну систему часу виконання. У ньому є звичні елементи, наприклад, типи даних (int, float і т.д.), структури, функції,

покажчики і керуючі конструкції (while, if ... else і оператор for). Також є доступ до функцій стандартної бібліотеки програмування С, наприклад, до тих, які оголошені в stdlib.h і stdio.h.

Також в Objective-C додана підтримка об'єктно орієнтованого програмування в стилі Smalltalk, тобто об'єктам надсилаються повідомлення замість виклику методу.

За аналогією з інтерфейсами в Java, в Objective C є протоколи. Протокол визначає набір селекторів, які повинен підтримувати об'єкт, який реалізує протокол. Протоколи повинні описувати методи, які реалізуються певним класом.

Основні завдання для яких вони використовуються:

- очікування, що клас підтримує протокол виконає описані в протоколі функції;

- підтримка протоколу на рівні об'єкта, не розкриваючи методи і реалізацію самого класу;

- з причини відсутності множинного спадкоємства - об'єднати спільні риси декількох класів.

Swift є більш молодим і, на даний момент, що стрімко розвивається мовою програмування. Він розробляється компанією Apple, як майбутня заміна Objective-C. Swift замислювався як швидкий і ефективний мову програмування з відгуком в реальному часі, який легко можна вставити в готовий код Objective-C. Тепер розробники можуть не тільки писати більш надійні і безпечні коди, але також економити час і створювати додатки з розширеними можливостями.

Swift бере досить багато з Objective-C, проте він визначається не покажчиками, а типами змінних, які обробляє компілятор. За аналогічним принципом працюють багато скриптові мови. У той же час, він надає розробникам багато функцій, які раніше були доступні в C ++ і Java, такі як визначаються найменування, так звані узагальнення і перевантаження (overloading) операторів.

Swift працює з фреймворками Cocoa і Cocoa Touch і сумісний з основною кодової базою Apple, написаної на Objective-C. Swift розробляється як більш безпечний мову в порівнянні з Objective-C, до того ж Swift, на відміну від Objective-C, легко читається мова, як і Python.

На даний момент ці мови взаємно доповнюють один одного, але основна частина додатків написана на Objective-C. [5]

1.6 Існуючі архітектури для мобільної розробки

Побудова правильної архітектури мобільного застосунку має першорядне значення для всього проекту. Приділивши належну увагу якості коду, масштабованості системи, вибору API сервера і надійних технологій, можна уникнути більшості проблем в подальшому.

Закладена програмістами архітектура безпосередньо впливає на те, чи знадобиться постійна підтримка і "ремонт" додатка або ж все буде працювати (і продовжувати працювати) як треба. Трапляється так, що бажання заслужити лояльність замовника або простий непрофесіоналізм менеджерів проекту вганяють команду в жорсткі рамки, не даючи можливості приділити потрібний час для того, щоб продумати архітектуру програми. Прямо протилежний результат не змусить себе чекати: купа багів, недоробок, витрат і "перезалівок". Тому необхідно все детально продумати ще на початковому етапі проекту. [6]

Отже, найважливішим аспектом розробки мобільного застосування є вибір правильного шаблону проектування. В першу чергу розглянемо особливості хорошою архітектури програми:

- збалансований розподіл функцій між діючими об'єктами;
- тестової програми;
- зручність використання.

В даний час існує багато варіантів архітектури шаблонів проектування, найбільш популярними з яких є:

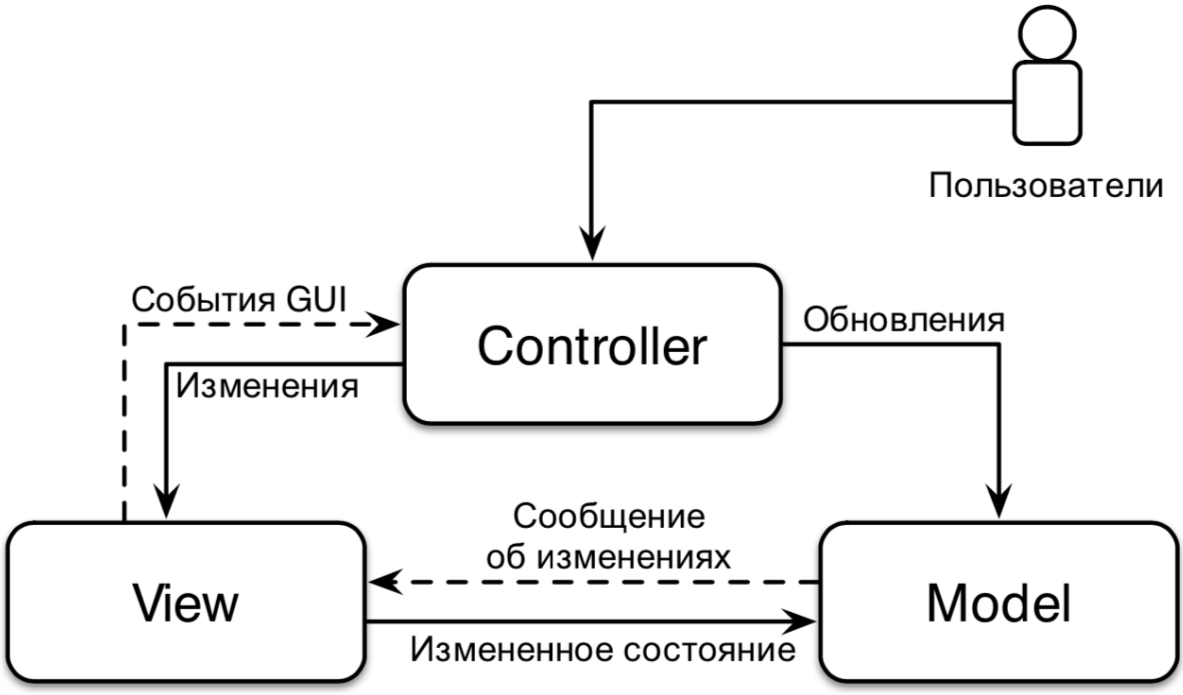
- Model-View-Controller (MVC);

- Model-View-Presenter (MVP);

- Model-View-ViewModel (MVVM).

Розглянемо кожен з них.

MVC - схема використання декількох модулів, за допомогою яких модель, користувацький інтерфейс і взаємодія з користувачем розділені на три окремих компонента таким чином, щоб модифікація одного з компонентів



надавала мінімальний вплив на інші. Традиційне уявлення MVC моделі

Рисунок 1.1 - Класична модель MVC

представлено на рисунку 1.1. [7]

Модель (Model) містить основні дані, наприклад, змінні, підключення до зовнішніх RSS-каналах або зображення, докладні функції і числову інформацію. Цей шар повністю відділяється від візуального оформлення, тому можна легко змінити вигляд дисплея, і це не вплине на дані.

Вид (View) відповідає за стиль відображення інформації на дисплеї. Як уявлення виступає екран з графічними елементами на якому, наприклад, можна змінити стиль або видаляти елементи.

Контролер (Controller) управляє запитами користувача і використовує модель для реалізації необхідної реакції.

Але традиційна MVC архітектура не може бути застосована до сучасної розробці додатків під мобільну ОС iOS, так як для забезпечення зв’язування модулів View та Model у фреймворці для iOS немає необхідних нативних інструментів, тому компанія Apple модернізувала даний шаблон проектування під свою архітектуру, яка називається Cocoa MVC (рисунок 1.2). Тут контролер (controller) є посередником між видом і моделлю, так що вони не взаємодіють один з одним.

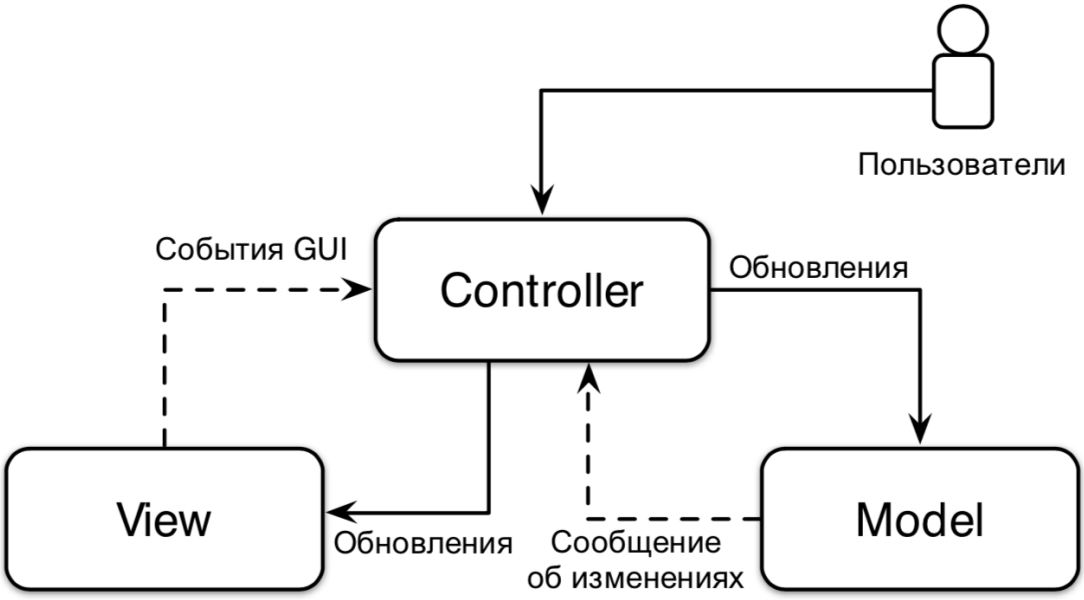


Рисунок 1.2 - Cocoa MVC модель

Особливістю цього шаблону проектування MVC є те, що контролер (controller) настільки залучений в життєвий цикл виду (view) за допомогою підкласу UIViewController, тому важко сказати, що вони дійсно розділені.

Обговоримо архітектуру з точки зору якостей, виділених вище:

- розподіл - вид (view) і модель (model) в даному випадку повністю розділені, але вигляд (view) і контролер (controller) тісно пов'язані між собою;

- тестованість - через неповний розподіл можна тестувати тільки модель (model);

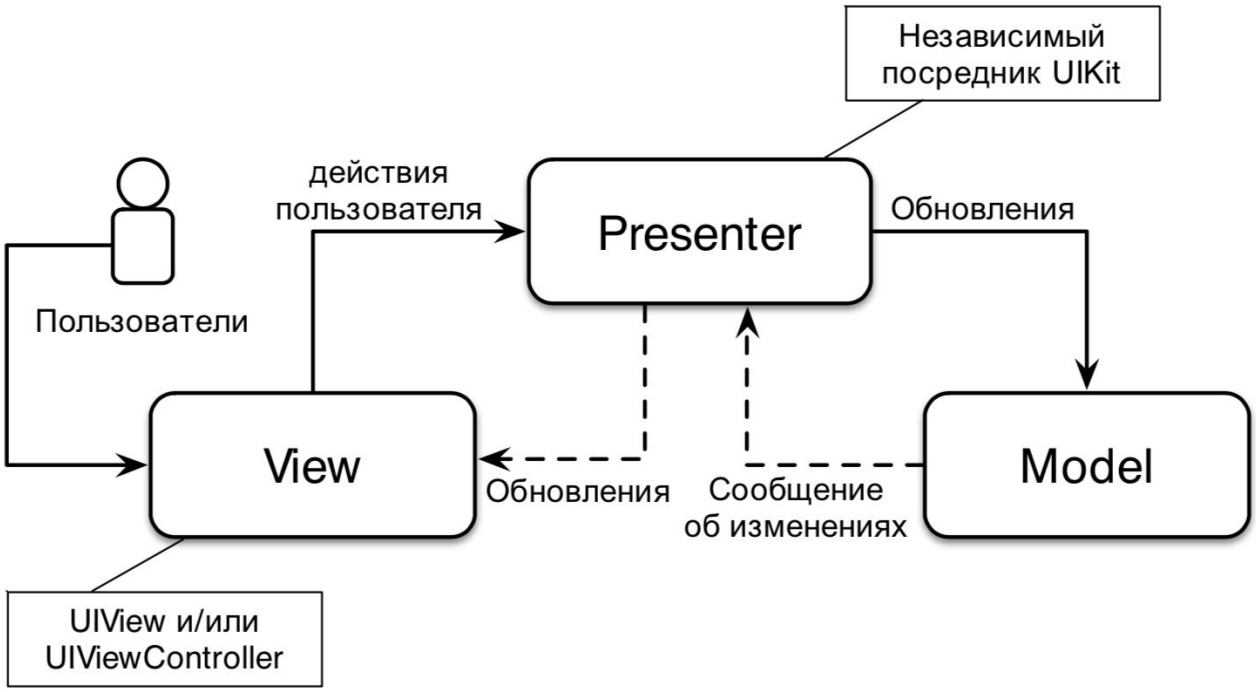
- простота використання - використовується менша кількість коду в порівнянні з іншими моделями, крім того є найпоширенішою моделлю.

Недоліки:

- неможливість тестування;

- контроллер стає дуже великим, наділений багатьма відповідальностями;

- неможливість перевикористовувати.



Cocoa MVC є найкращим шаблоном проектування з точки зору простоти і швидкої реалізації.

Розглянемо наступну архітектуру. MVP є шаблоном проектування похідним від MVC (рисунок 1.3). Основним завданням MVP є зробити view повторно використовуваним.

Рисунок 1.3 - MVP модель

Presenter, як і контролер (controller) управляє видом (view) і моделлю (model), але при цьому вид (view) кожен раз не перестворюється.

Підкласи UIViewController фактично належать виду (view), а не presenter і ця різниця забезпечує чудову тестованість.

Розглянемо коротко особливості MVP:

- розподіл - більша частина обов'язків розділена між presenter і моделлю (model);

- можливість перевикористання;

- тестованість - можна перевірити більшу частину бізнес-логіки;

- простота використання - кількість коду значно збільшиться в порівнянні з MVC моделлю, але в той же час, ідея MVP досить проста.

В результаті шаблон проектування MVP означає чудову тестованого і велика кількість коду.

Недоліки:

- presenter всеодно стає великим;

- треба писати більше коду.

MVC і MVP парадигми дуже схожі один на одного, але їх застосування залежить від умов використання. Для MVC - це там, де вид (view) оновлюється кожного разу по якій-небудь події, а для MVP, коли вид (view) не потрібно щоразу перестворювати. [8]

Розглянемо останній з обраних шаблонів проектування. MVVM з'явився для обходу обмежень патернів MVC і MVP, і об'єднує деякі з їх сильних сторін. Ця модель вперше з'явилася в складі фреймворка Small Talk в 80-х, і була пізніше покращена з урахуванням оновленої моделі MVP. Схема роботи MVVM представлена на рисунку 1.4.

Вид (view) і модель (model) вже були розібрані, але в даному випадку посередником виступає модуль view model.

View model готує всі дані, які необхідно відобразити на екрані. View model Він ніколи не посилається на вигляд (view), замість цього він постійно стежить за будь-якими змінами view model, і як тільки зміни відбудуться, вони відразу будуть відображені на екрані.

Знову повернемося до оцінки шаблону проектування:

- розподіл - по суті, вид (view) MVVM має більше обов'язків, ніж вид (view) MVP, оскільки перший шаблон оновлює стан системи з точки зору моделі (model), коли другий направляє всі події до Presenter і не оновлює себе;

- тестованість - повний розподіл класів дозволяє легко проводити модульні тести;

- простота у використанні - обсяг коду теж більше, ніж в MVC. Але в порівнянні з MVP код більш лаконічний так, як не потрібно спрямувати всі події з виду (view) до presenter і оновлювати його вручну.

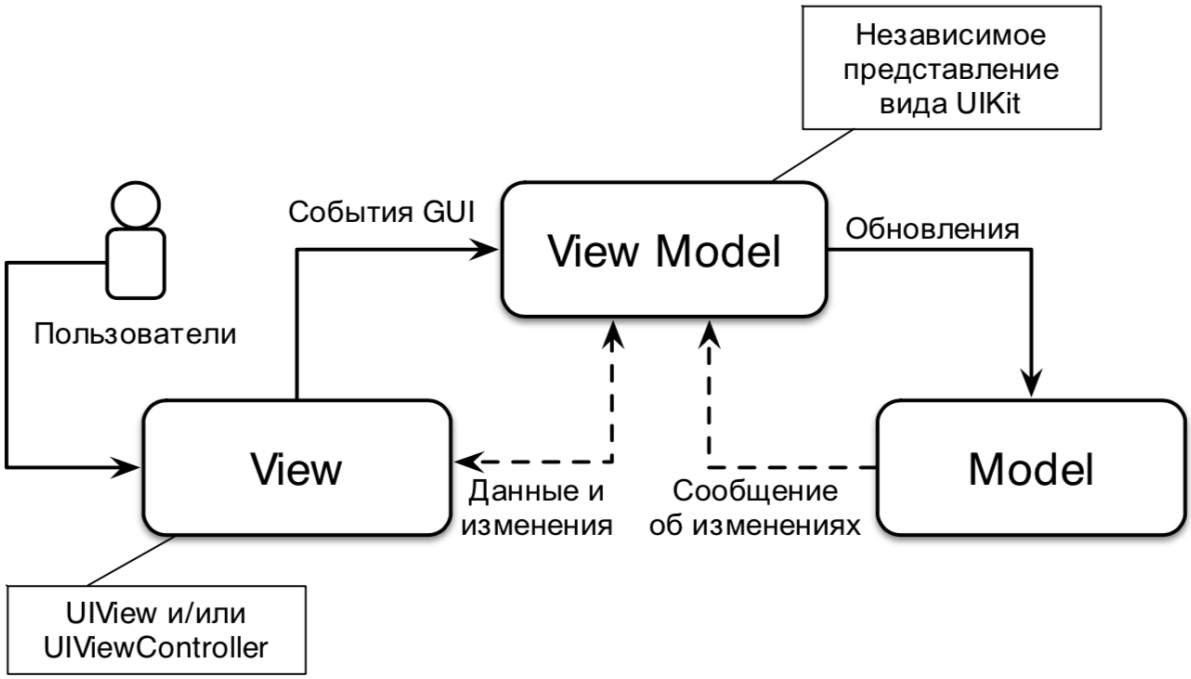


Рисунок 1.4 - Модель MVVM

Тому MVVM є дуже привабливим варіантом, так як він поєднує в собі переваги вищезазначених підходів, і, крім того, він не вимагає додаткового коду для виду (view) та тестованість знаходиться на хорошому рівні. Але і MVVM справді має свої проблеми. Він дає пояснення до розподілу обов'язків дуже високго рівня (абстрактно).

Недоліки:

- оскільки спочатку MVVM розроблявся не для мобільних застосунків, а для десктопних застосунків (MVVM був винайдений розробниками компанії Microsoft), то у розробників мобільних застосунків виникло багато непорозумінь щодо цієї моделі, бо кожен розробник інтерпретує реалізацію по-своєму;

- необхідність застосування "третіх" бібліотек;

- складність розуміння реативного підходу, складність дебагу.[9]

1.7 Висновки до розділу

Кожний розробник обирає саме ту архітектуру, яка йому зручніша і яка відповідє вимогам поставлених задач. У багатьох випадках мобільні застосунки розробляються для вирішення бізнесових задач і в свою чергу для бізнеса важливі часові дедлайни, швидкість та якість. Зазвичай ці мобільні застосунки є частиною клієнт-серверноюї архітектури, де клієнтом виступає як раз мобільний застосунок. Такий тип застосунків є найпопулярнішим і тому саме для цього типу є сенс і потреба у створенні дійсно ефективного й простого шаблону (архітектури). Також головною вимогою від бізнесу є можливість функціонування мобільного застосунку в офлайн-режимі.

У кожної із розглянутих архітектур є свої переваги і недоліки, тому кожний наступний новий підхід намагається подолати недоліки своїх попередників, тим самим провокуючи створення ще нових недоліків. Також основною проблемою є те, що всі підходи описані дуже поверхнево і це призводить до того, що кожний розробник інтерпретує визначення архітектур по-своєму і використовує свої підходи. Це, в свою чергу, приводить до створення багатьох різних реалізацій, які частково або зовсім не відповідають початковій задумці. Отже, було запропоновано створити новий архітектурний підхід на основі всіх трьох розглянути архітектурних підходах та головною метою якого буде використання переваг кожної з попередніх. Нова архітектура ставить більш жорсткі рамки до конкретизації, а також буде запропонований конкретний приклад реалізації.

2 КОНЦЕПТ НОВОЇ АРХІТЕКТУРИ

2.1 Задачі Нової Архітектури

Суть нової архітектури полягає в тому, щоб використати усі переваги існуючих і запропонованих раніше архітектур та покрити необхідні бізнесові аспекти розробки: швидкість, якість, можливість розширення.

Для цього розглянемо детальніше переваги та недоліки існуючих архітектур та оберемо найкащі особливості кожної з них (рисунок 2.1).

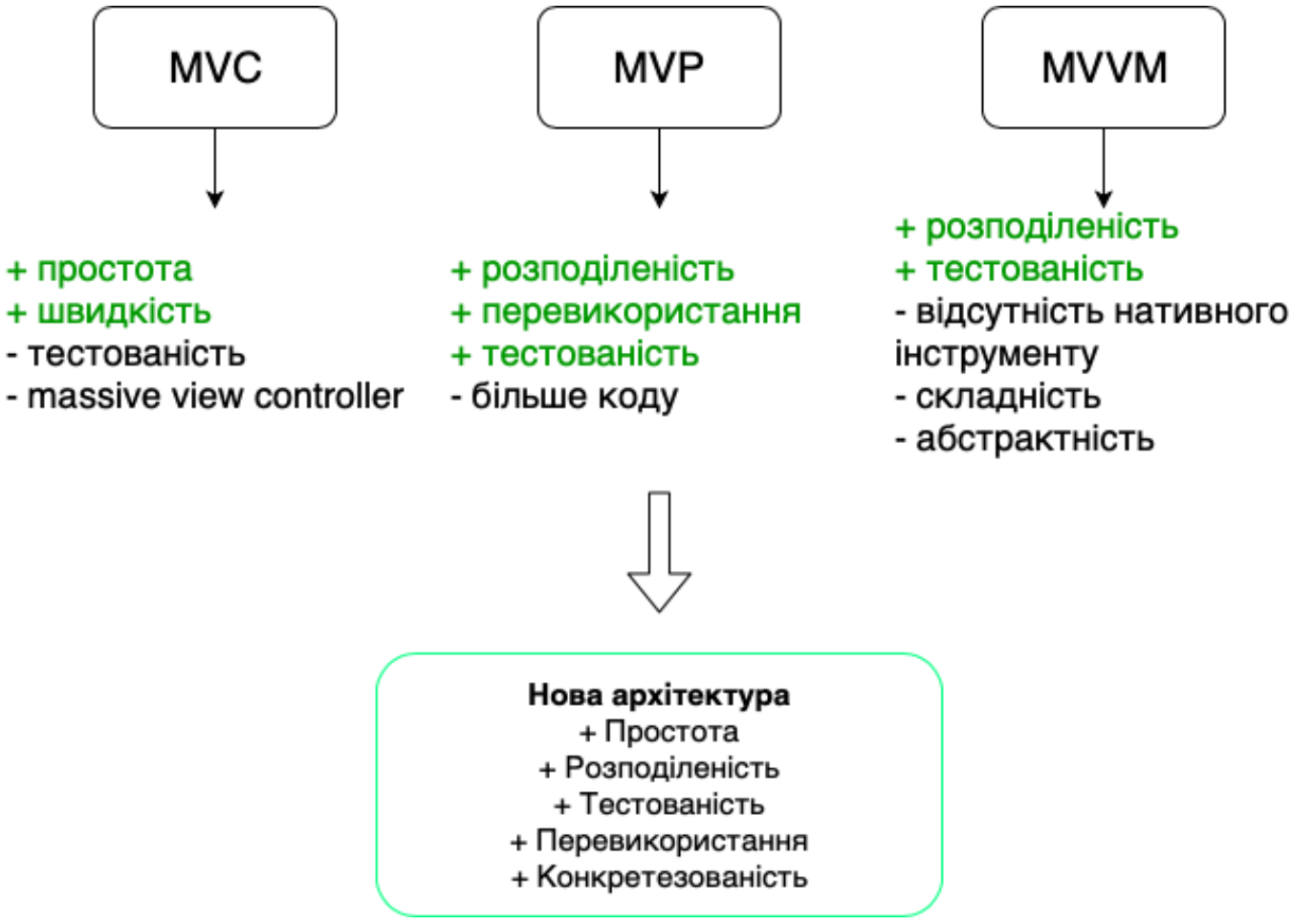


Рисунок 2.1 - Переваги і недоліки архітектур

Отже, для нової архітектури були обрані основні особливості для досягнення максимально ефективного результату:

- Розподіленість

Найважливішим пунктом для нової архітектури є розподілені відповідальності між модулями. Архітектури MVP і MVVM прописують відповідальності для кожного модуля, але все ж з MVP з часом все одно буде мати найбільш відповідальний і найбільший по розміру модуль - Presenter.

Також основною проблемою багатьох розробників-початківців використовують архітектуру MVC і то, не завжди правильно. Дуже часто виходить так, що при розширенні та ускладненні функціональності один модуль (ViewController) вміщає в собі всю логіку застосунку. Такий підхід серед розробників називають “Massive View Controller” (у перекладі з англійської - масивний View Controller). Ось і головний недолік MVC - відсутність розподілу відповідальності між модулями.

- Тестованість

Покриття тестами не тільки важливе для якості продукту, а й дуже часто є бізнес умовою. Наприклад, замовники часто вимагають від застосунку покриття тестами від 70% усього коду. Із архітектурою MVC цього досягти неможливо, бо покрити тестами можна тільки модуль Модель. Абстрагованість модулів у MVP і MVVM дозволяє зробити максимальне покриття тестами можливим.

- Перевикористання

Особливість перевикористання означає зменшення дублювання коду і максимально перевикористовувати вже існуючі компоненти. Ця можливість є дуже важливою на етапах створення достатньо комплексного застосунку, а також на етапі розширення функціональності застосунку. З архітектурою MVC доводиться кожен раз писати заново, а з архітектурою MVP є можливість перевикористати модуль ще раз та підставити його на нове місце.

- Конкретизованість

Цей пункт не є перевагою жодної з архітектур, бо, насправді, це є недоліком багатьох архітектур, зокрема архітектури MVVM. Проблема абстрагованого визначення архітектури MVVM та не призначеного для iOS розробки підходу дає свої плоди - непорозуміння розробників щодо способу

реалізації. Тому однією з головних ідей є конкретезовані способи реалізації підходів, зазначених в описі нової архітектури.

- Простота

Простота це одна із найголовніших особливостей архітекутри MVC, але і в той самий час є головним недоліком архітектури MVVM. У архітектури MVVM дуже високий “поріг входження”. Тому задача для нової архітектури стоїть в тому, щоб навіть для простого застосунку із самого початку розробки було легко і зрозуміло застосовувати нову архітектуру та подальше будувати нові можливості із задоволенням.

2.2 Архітектура

Розглянемо основну концепцію нової архітектури та взаємозв’язки між модулями (схема наведена у додатку А).

Головними модулями архітектури є:

1. View

Як і в будь-якій іншій архітектурі View є необхідним модулем для будь-якого мобільного застосунку. Саме цей модуль включає користувацький інтерфейс: він відображає дані і саме він взаємодіє з користувачем (рис. 2.3). Але View не є відповідальним за те, де дані взяти і в який вигляд їх треба привести для відображення, а також він не є відповідальним за те, як йому обробляти події від взаємодії з користувачем. Цей моділь відповідальний за наступні дії: як відобразити підготовлені дані на екрані (мається на увазі графічно), за реагування подій з екрану, а також він знає до кого зветатись за даними і кому віддавати відповідальність за оброрбку користувацьких подій.

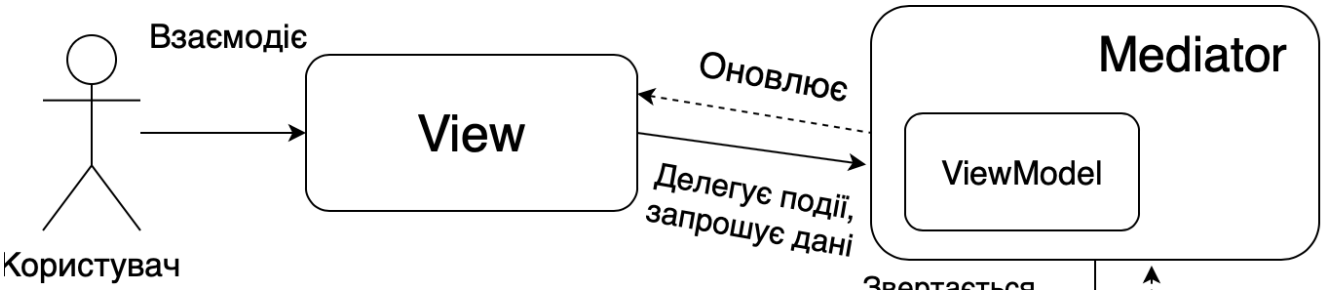


Рисунок 2.3 - Схема View

Отже, спілкується тільки з користувачем та Mediator, якак як раз і прриймає делеговані запити від View.

2. Mediator

Найголовнішим модулем архітектури є модуль Mediator, який виступає головним обробником даних і є відповідальним за бізнес логіку. Спілкується модуль із View не на пряму, він просто виступає об’єктом підготовлених для відображення даних і таким чином View сама звертається до Mediator за потрібними даними. Також Mediator має достатньо тісні зв’язки (рис. 2.4) із Model.

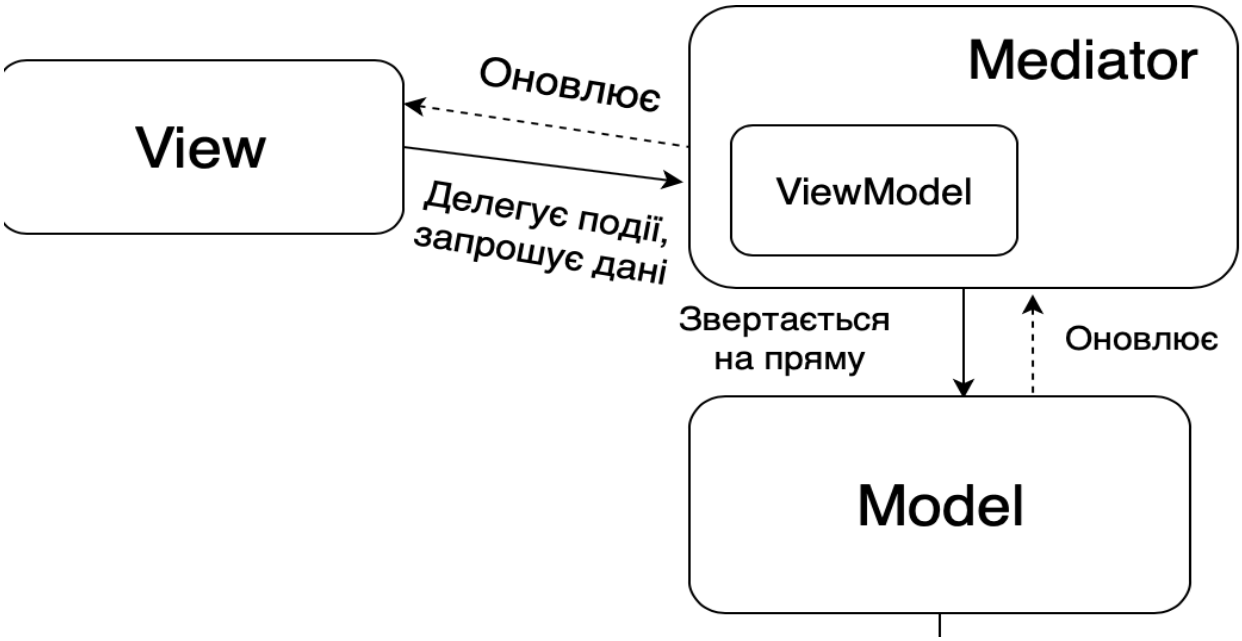


Рисунок 2.4 - Модуль Mediator

Так як Mediator не тільки підготовляє дані для відображення, а ще й обробляє запити від View, то, звичайно, Mediator для цього працює об’єктами Model. Цей рівень є останнім перед View де можна працювати з об’єктами Моделі напряму. Наприклад, на рівні View це порушення відповідальностей.

У випадку, якщо модель змінюється сама, то модуль Model оповіщає о змінах Mediator і вже Mediator сама вирішує як саме “приймати” оновлення та як їх обробляти.

3. Model

Модель - це рівень, який відповідає за дані. Відповідає за те де їх взяти, де їх зберегти, в якому вигляді створити, тощо. Ідея “офлайності” є не тільки частою вимогою від бізнесу, а й взагалі дуже ефективним інструментом для розробки. Режим офлайну досягається завдяки створенню локальної бази даних у мобільному застосунку. Це дозволяє підтримувати ідею архітектури - розподіленість.

Model зв’язана із Mediator “каналом для обновлень” - Model може тільки відправляти повідомлення (рисунок 2.5). Через Service дані завантажуються із сервера, а потім ці дані зберігаються у Базі Даних. Після збереження даних Model повідомляє ViewModel про зміни в модулі.

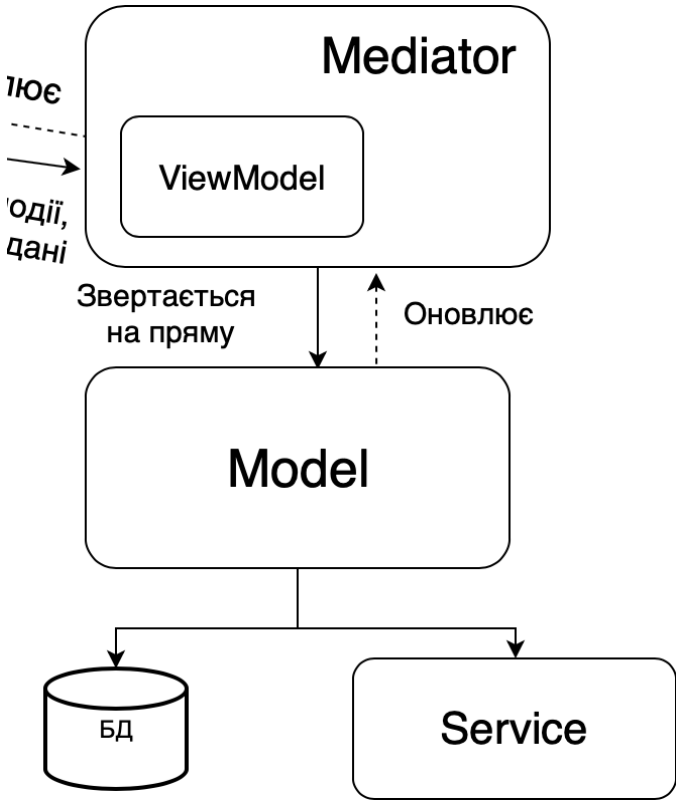


Рисунок 2.5 - Схема Model

2.3 Детальний опис модулів

Абстрактне розуміння архітектури та взаємозв’язків між модулями вже сформовано і тепер потрідно спускатись рівнем нижче - тож наступним етапом буде безпосередньо опис кожного із модулів.

Для більш чіткішого розуміння архітектури необхідно детально розглянути кожний із модулів та його мету, відповідальність, приклади реалізації.

1. View

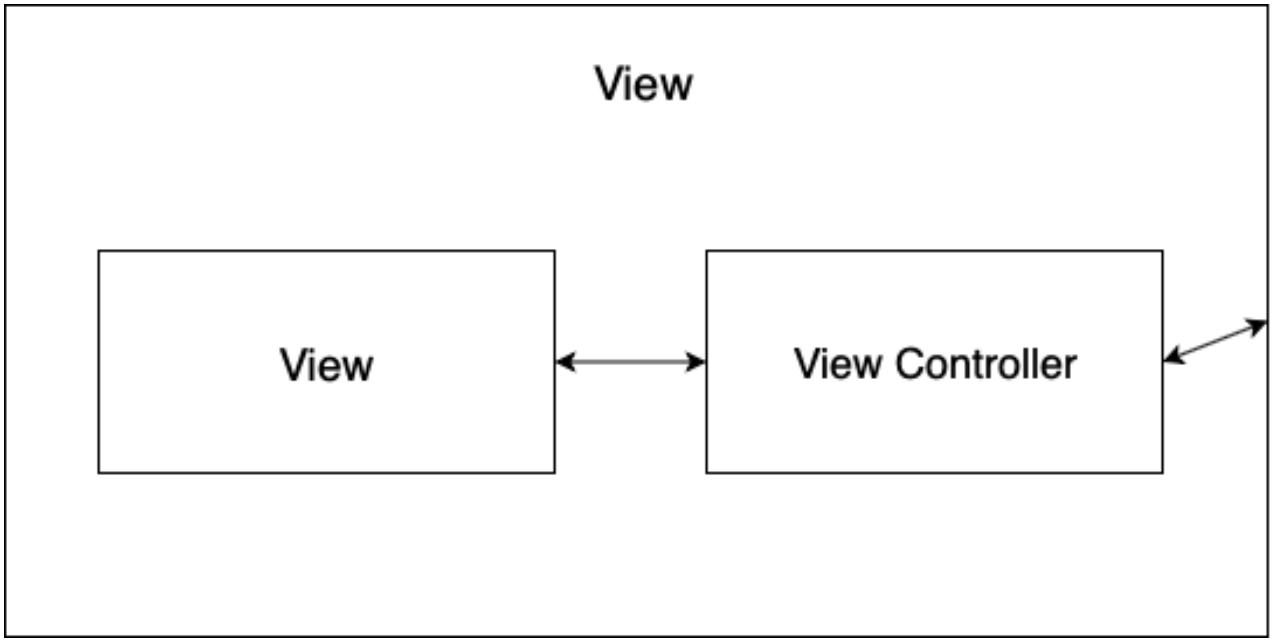


Рисунок 2.6 - Модуль View

У розробці під iOS модуль View (рисунок 2.6) включає в себе 2 компоненти: view і view controller. Ці дві компоненти відносяться конкретно до технічної сторони iOS розробки.

Компонента View - це компонента, яка головним чином відповідає за користувацький інтерфейс, а саме за те, що буде відображено на екрані. View може бути створено завдяки вбудованому у XCode інструменту, який називається Interface Builder або ж може бути створений у коді. Зручніше, звичайно, створювати UI у Interface Builder, так як це візуальний інструмент, в якому можна використовуючи базові UI елементи на кшталт Кнопка, Напис, Зображення, Таблиця, тощо. Всі розміри та автоматичний макет можна налаштувати саме там. Якщо ж для розробки необхідно створювати більш

складні та кастомізовані елементи, то, зазвичай, звертаються до способу створення UI елементів в коді. Але, зрозуміло, що цей спосіб є менш наглядним.

Отже, компоненти View несуть мінімум відповідальності, та є спроможними тількі на те, щоб відмалювати себе та прорахувати автоматичний макет.

Не менш важливим елементом є компонента View Controller. Ця компонента є специфічна для платформи iOS. В багатьох випадках вона дуже тісно пов’язана з View, так як основними задачами View Controller’а є контроль життєвого циклу View і відображенням View в різних орієнтаціях пристрою. Отже, View просто делегує всі свої запити, а View Controller вже обробляє ці запити і відповідає за те, якими даними наповнити View та як відобразити в конкретних випадках View. Ці дві компоненти завжди тісно зв’язані саме тому, що View Controller дуже часто спілкується зі View і навпаки: view controller каже які дані треба відобразити, а View повідомляє контролеру про всі взаємодії користувача з UI.

Часто вважають, що саме компонента ViewController має тримати усю логіку застосунку і це можна пояснити, так як у контролера є доступ і до View, і до будь-якого іншого модуля, наприклад, до модуля сервіс. Такий підхід як раз називається MVC - коли View Controller виступає модулем Controller, а компонента View сама по собі є модулем View. Але у такого підходу багато мінусів саме із-за великої кількості відповідельностей та навантаження.

Отже, в новій архітектурі було вирішено виокремити пару View-ViewController в окремий модуль View, тому що основна ціль модуля View - відображення користувацього інтерфейсу та взаємодія з ним. View Controller не має містити в собі логіку стосовно того, де і які дані брати, як ними маніпулювати, та й тем більш контроллер не повинен знати про модулі Service та Model. Він спілкується тільки з одним модулем - Mediator, від якого він отримує дані у самому примітивному вигляді, а також делегує йому логіку обробки користувацьких дій. Спілкування між цими модулями побудовано

абстрактно - на основі протоколів з методами. Наприклад, контролер запрошує текст, який потім відображує у лівому верхньому кутку, і йому абсолютно всеодно де ті дані бере Mediator, головне - це отримати цей текст. Також, контролер повідомляє Mediator про те, що користувач натиснув кнопку, а далі, що робити з цією подією - це вже не відповідальність ViewController'а. Саме завдяки виокремленню таких відповідальностей, View може бути перевикористана. Вона може працювати з будь-яким іншим Mediator, і завдяки цьому мати інші дані, іншу функціональність, іншу ідею. Тобто, єдина відповідальність модуля View - це відображення даних, які отримуються від Mediator.

2. Mediator

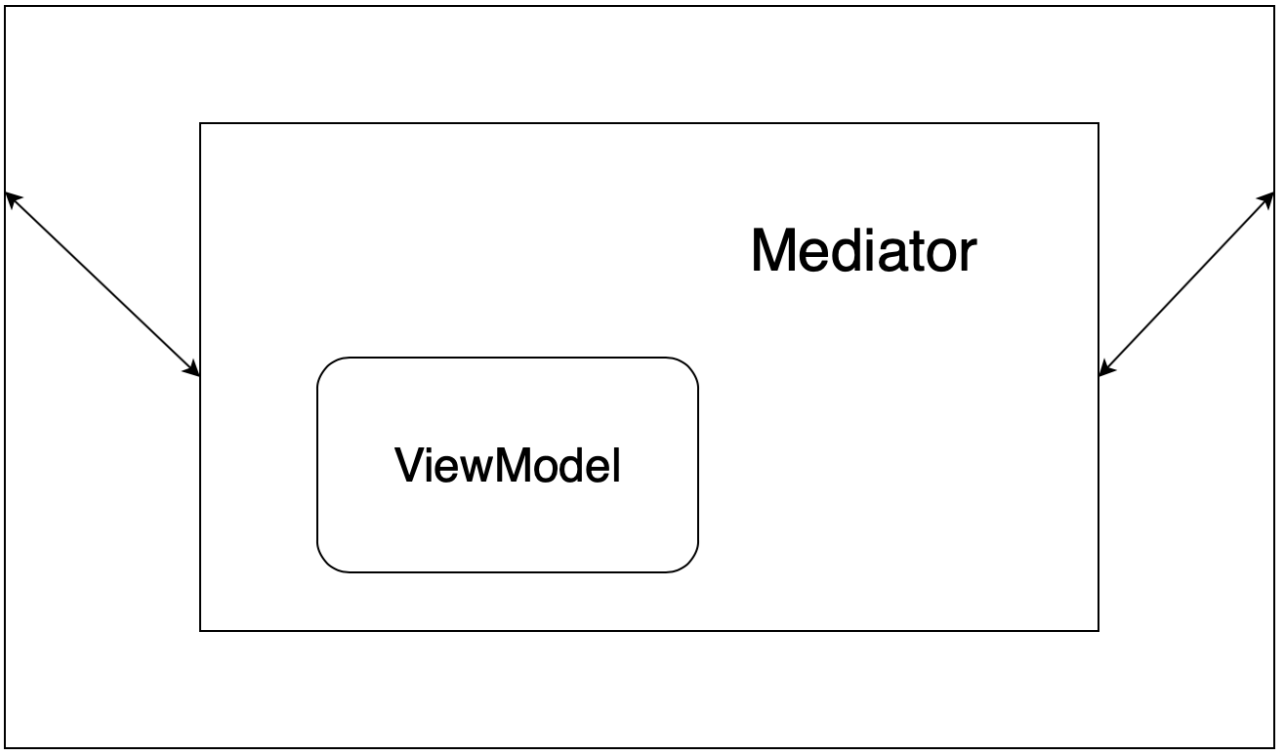


Рисунок 2.7 - Модуль Mediator

Головною компонентою в модулі Mediator (рисунок 2.7) є Mediator, який створює ViewModel'і, що виступають примітивним виглядом даних для View. За всю бізнес логіку відповідає саме цей модуль.

Іншими словами, Mediator - з одного боку, абстракція View, а з іншого - обгортка даних з Model, що підлягають зв'язанню. Тобто, вона містить Model,

перетворену для View, а також команди, якими може користуватися View, щоб впливати на Model. Mediator сам спілкується з моделлю, тобто було знято з контролера цей обов'язок і тепер контролер займається тим, чим треба - він працює з View і навіть не знає про існування моделі.

Mediator виступає посередником між View та Model, тобто він працює напряму з обома модулями. Для більшого розуміння розглянемо конкретний приклад спілкування: view повинна відобразити дані, вона просить ці дані у Mediator, Mediator у свою чергу запитує дані у Model, отримує їх, перетворює їх у потрібний вигляд для View і віддає їх у вигляді ViewModel'і до View. ViewModel - це примітивна структура даних, яка просто містить готові для відображення дані. Наприклад, ViewModel може містити заголовок, значення, тип валідації, тощо.

Також є інша додаткова особливість у Mediator - це можливість перевикористання цієї компоненти. Кожна View містить одну Mediator, але ViewModel не підв'язується до конкретної реалізації View - вони спілкуються абстрактно. Тому завдяки можливості компонування пар View і ViewModel можна створити так званий "констуктор" - створити базові модулі (модулі View і модулі ViewModel) і компонувати ці пари задля досягнення потрібного функціоналу.

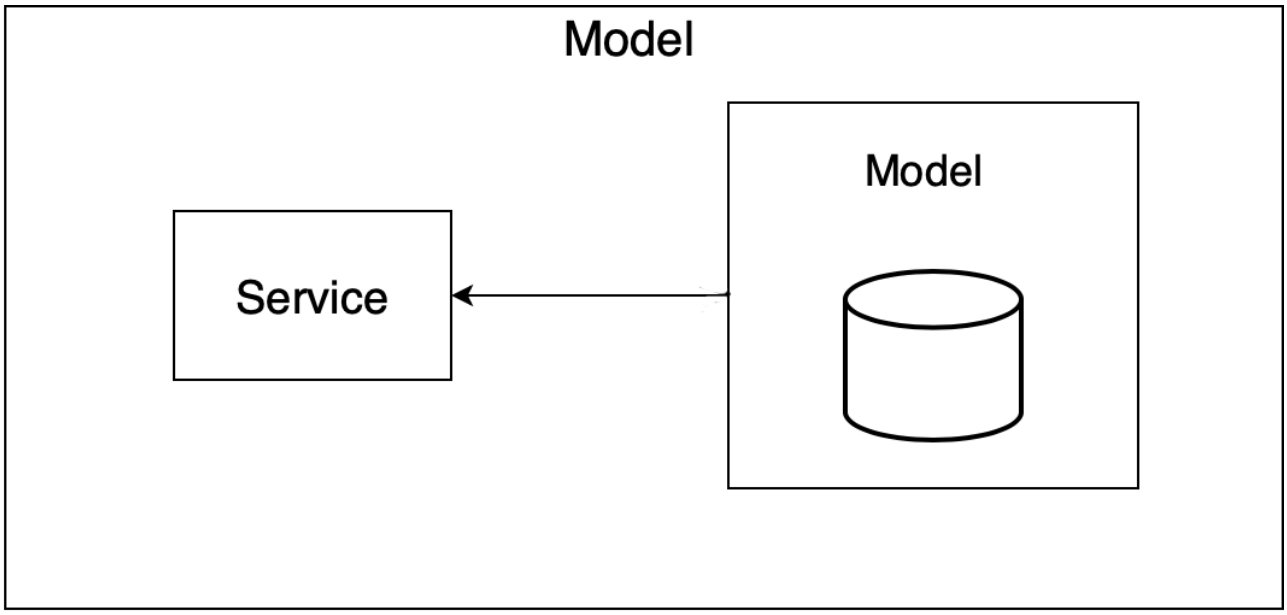
Отже, основною метою Mediator є роль посередника між Model та View, а саме: маючи бізнес логіку Mediator перетворює об'єкти з модуля Model в потрібний стан та вигляд і передає готові дані View. Також Mediator є віповідальним за обробку взаємодії з користувачем: саме Mediator повідомляє що View повинна зробити і як повинна відреагувати на ту чи іншу дію користувача. Тобто ідея Mediator у тому, щоб максимально розвантажити ViewController та якнайбільше абстрагувати його від бізнес логіки та модуля Model.

3. Model

Модуль Model відповідає за рівень фундаментальних даних, забезпечуючи майже повну відповідність рівню моделі на стороні backend.

Зазвичай, на цьому рівні також відбувається маніпулювання даними, наприклад: завантаження, збереження, відправка, тощо. Саме тому в реалізації нової архітектури використовуються дві компоненти (рисунок 2.8): та, що відповідає за представлення даних у системі і та, що проводить ці маніпулювання з даними (іншими словами - сервіс).

Рисунок 2.8 - Модуль Model



Модель представляє собою схему об’єктів та їх взаємозв’язки між собоюю Основною особливістю реалізації модуля Model в новії архітектурі є те, що модель завжди повинна зберігатись у базі даних для того, щоб забезпечити гнучкість спілкування модулів між собою. Перевагою такого підхода є те, що будь-які необхідні об’єкти можна запросити у будь-якому місці коду - це зручно тому, що не треба передавати із одного модуля в інший необхідні дані, та й не потрібно знати що потребує інший, абсолютно незалежний модуль. Кожен модуль сам може запросити необхідні йому дані. Також завдяки цьому забезпечується абстрагованість компонент і можливість перевикористання.

Модель вміє себе зберігати, завантажувати та відправляти куди треба і це завдяки компоненті Service (сервіс). Сервіс відповідає за всі можливі методи маніпулювання об’єктами, а також є відповідальний за спілкування системи із зовнішніми сервісами (наприклад, backend). Отже, за всіма необхідними діями

Модель звертається до свого сервісу, іншими словами можна сказати, що модель делегує завдання по маніпуляції саме сервісу, і вже після виконання необхідниї задач сервіс звітує Моделі про статус виконаних дій.

З Моделлю спілкується тільки Mediator. Mediator може запитувати будь-які необхідні дані, незважаючи на те, де їх завантажувати, в якому вигляді, тощо, а Модель буде вже сама вирішувати де і як їх завантажувати: чи то з локальної бази даних, чи із серверу. Приклад спілкування може бути таким: Mediator запитує всі записи об’єкта Person, модель спочатку перевіряє, чи є об’єкти у локальній базі, якщо є, то негайно повертає їх, та далі при наявності мобільного з’єднання все одно звертається до сервісу з проханням завантажити дані із серверу. Сервер завантажує дані та повертає їх моделі, модель зберігає нові записи у локальну базу. Далі Модель вирішує чи необхідно повідомити про нові дані Mediator чи ні.

Отже, сама компонента Model представляє собою набір описаних об’єктів (які майже співпадають з моделями на стороні сервера) та взаємозв’язків між ними. Модель може сама себе зберігати у локальну базу, а також діставати дані. Сервіс є відповідальним за те, щоб спілкуватись із сервером - встановлювати з’єднання, відправляти дані, отримувати дані. Основна мета Model є забезпечення маніпулювання найактуальнішими записами.

2.4 Висновки до розділу

Отже, були проаналізовані існуючі архітектури та були виділені основні недоліки та переваги кожної з них. На основі сформованих переваг та недоліків були обрані ключові особливості для Нової Архітектури: простота, розподіленість, тестованість, конкретизованість та перевикористання. На основі сформованих задач-особливостей було створено та запропоновано нову архітектру.

В даному розділі були детально описані основні компоненти Нової Архітектури. Також були пояснені взаємозв’язки між компонентами. Для

кращого розуміння компонент були наведені приклади різних сценаріїв роботи застосунків.

3 ПРАКТИЧНЕ РІШЕННЯ

Для демонстрації прикладу застосування Нової Архітектури на практиці оберемо ідею та тему для майбутнього мобільного застосунку. Для найкращої демонстрації буде обрано рішення для стартапу (або малого бізнесу) у сфері продажу, а саме це буде застосунок для ринку Об’єднаних Штатів Америки, в якому зібрано знижки та акції усіх компаній, які співпрацюють із компанією замовника. Розробка застосунків такого типу є дуже поширена серед iOS розробників і користується великим попитом у всіх українських аутсорс компаніях. Такий застосунок, зазвичай, розробляється десь пів року приблизно двома розробниками. Нова Архітектура як раз чудово підходить для реалізації цього застосунку, так як призначена для реалізації середніх по об’єму застосунків з обмеженими сроками, а також зазначається, що у майбутньому розроблений застосунок буде підтримувати інша команда, тому за рахунок використання Нової Архітектури забезпечується простота реалізації та можливість безпроблемного розширення функціоналу.

Отже, головною метою при демонстрації прикладу реалізації Нової Архітектури є зосередження уваги на ключових моментах (наприклад, місця, де явно помітні переваги архітектури) та зазначення рекомендованих підходів, способів реалізації (якщо їх декілька) та бібліотек. Також, необхідно зазначити, що для досягнення максимальної ефективності буде використана мова програмування Swift.

Перед початком розробки поділимо процес реалізації Нової Архітектури на наступні етапи:

1. Реалізація Model, ініціалізація бази даних.
2. Реалізація Service.
3. Реалізація базових класів для View і Mediator.
4. Реалізація конкретних екранів (конкретних View і Mediator).

3.1 Реалізація Model, ініціалізація бази даних

Найпершим етапом при реалізаціїї Нової Архітектури є підготовка рівня Моделі, так як саме Модель забезпечує базовий фундамент для подальшої логіки та подальшого розвитку застосунку. Але, бувають випадки, коли сторона backend ще не готова і iOS розробнику ще не відомо які сутності будуть використовуватись, то в такому випадку припускається можливим розпочати розробку з View рівня - тобто розробляти спочатку суто UI компоненти з абстрактними зв’язками до Mediator.

Для досягнення максимальної ефективності Нової Архітектури пропонується в якості локальної бази даних використовувати нативну бібліотеку Core Data. Використання саме нативних бібліотек та інструментів дозволить домогтись максимального рівня простоти та не завантажувати застосунок важкими, ненативними бібліотеками (у випадку, звичайно, якщо є аналог від Apple).

Core Data - це фреймворк, який управляє і зберігає дані в додатку. Можна розглядати Core Data, як оболонку над фізичним реляційним сховищем, що представляє дані у вигляді об'єктів, при цьому сама Core Data не є базою даних. Core Data - це фреймворк, який використовується для управління об'єктами модельного рівня у програмі. Він надає узагальнені та автоматизовані рішення загальних завдань, пов’язаних із життєвим циклом об'єкта та керуванням графіком об'єктів, включаючи постійність (persistence).[10]

Core Data зазвичай зменшє на 50-70 відсотків кількість коду, який пишеться для підтримки модельного рівня.[11] В першу чергу це пов'язано з такими вбудованими функціями, які не доведеться впроваджувати, тестувати чи оптимізувати:

- Відстеження змін та вбудованого керування undo та redo без ніякого редагування тексту.

- Підтримка розповсюдження змін, включаючи підтримку узгодженості зв’язків між об'єктами.

- Ледаче завантаження об'єктів, частково матеріалізовані ф'ючерси (несправні) та копіювання даних під час запису (copy-on-write), щоб зменшити накладні витрати.

- Автоматична валідація значень властивостей. Об’єкти Core Data розширюють стандартні методи перевірки кодування ключа-значення, щоб гарантувати, що окремі значення лежать у допустимих діапазонах, так що комбінації значень мають сенс.

- Інструменти міграції схем, що спрощують проведення змін схеми та дозволяють виконувати ефективну міграцію схеми на місці.

- Можлива інтеграція з контроллерами для підтримки синхронізації користувацького інтерфейсу.

- Групування, фільтрація та впорядкування даних в пам'яті та в інтерфейсі користувача.

- Автоматична підтримка зберігання об'єктів у зовнішніх сховищах даних.

- Вибір складних запитів. Замість того, щоб писати SQL, можна створювати складні запити, асоціюючи об'єкт NSPredicate із запитом на отримання.

- Відстеження версій та блокування для підтримки автоматичного багаторазового вирішення конфліктів.

- Ефективна інтеграція з ланцюжками інструментів macOS та iOS.

Отже, використання Core Data в Новій Архітектурі дозволить нам спроектувати схему сутностей та керувати об’єктами (діставати, зберігати, змінювати) з будь-якої точки коду, а це означає, що тепер не потрібно прокидувати необхідні об’єкти безпосредньо через контролери та інші класи.

Apple пропонує дуже корисний та зручний інструмент для роботи с Core Data, де можна візуально працювати з об’єктами та їх взаємозв’язками і тому саме з нього потрібно розпочати роботу. Необхідно схематично відтворити всі сутності рівня Model, встановити поля кожної з моделі та добавити зв’язки між

сутностями. Для прикладу, створимо необхідні об’єкти (схема наведена у додатку А).

Після налаштування Core Data у проекті, створення сутностей зі властивостями та додавання взаємозв’яків можна вважати, що етап реалізації Моделі та ініціалізації бази даних виконаний.

3.2 Реалізація Service

Обов’язково після створення моделі наступним етапом завжди буде налаштування сервісу (Service). Сервіс є відправною точкою для зв’язку так званим “зовнішнім світом” - backend. За допомогою Сервісу відбувається встановлення зв’язку зі сервером, а також ообмін даними по мережі. Також Service повинен бути відповідальним за обробку помилок (рисунки 3.2-3.3): можуть оброблятись помилки із серверу або створюватись свої помилки, які будуть відправлятись далі по ланцюжку відповідальності.

```swift
import Foundation

internal enum ServerError: ServerErrorProtocol {

    case noDataInResponse
    case noResponseCodeInResponse
    case wrongVerificationCode

    case customError(message: Any)

    var errorDescription: String {

        switch self {
        case .customError(let message):
            return "\(message)"
        default:
            return "\(self)"
        }
    }
}
```

```swift
internal enum UserServerError: ServerErrorProtocol {

    case noUserDataInResponse
    case noTokenInResponse
    case noSuccessCode
}

internal enum LocationError: Error {
    case noLocationAvailable
}

internal enum TimeoutError: ServerErrorProtocol {

    case timeoutReached

    var errorDescription: String {
        switch self {
        case .timeoutReached:
            return "Timeout was reached"
        }
    }

}

internal enum SocialNetworkError: ServerErrorProtocol {

    case unavailable

    var errorDescription: String {
        return "Service is currently unavailable"
    }

}
```

Рисунки 3.2, 3.3 - Кастомні помилки

Сервіс реалізовується за допомогою класів. Після створення необхідних методів для завантаження даними Модель повинна розширитись, щоб мати прямий доступ до сервісу. Це може бути забезпечено завдяки розширення (extension) до класів моделей. Для кожної сутності створюється кожний окремий файл для роботи із сервером (рисунок 3.4).

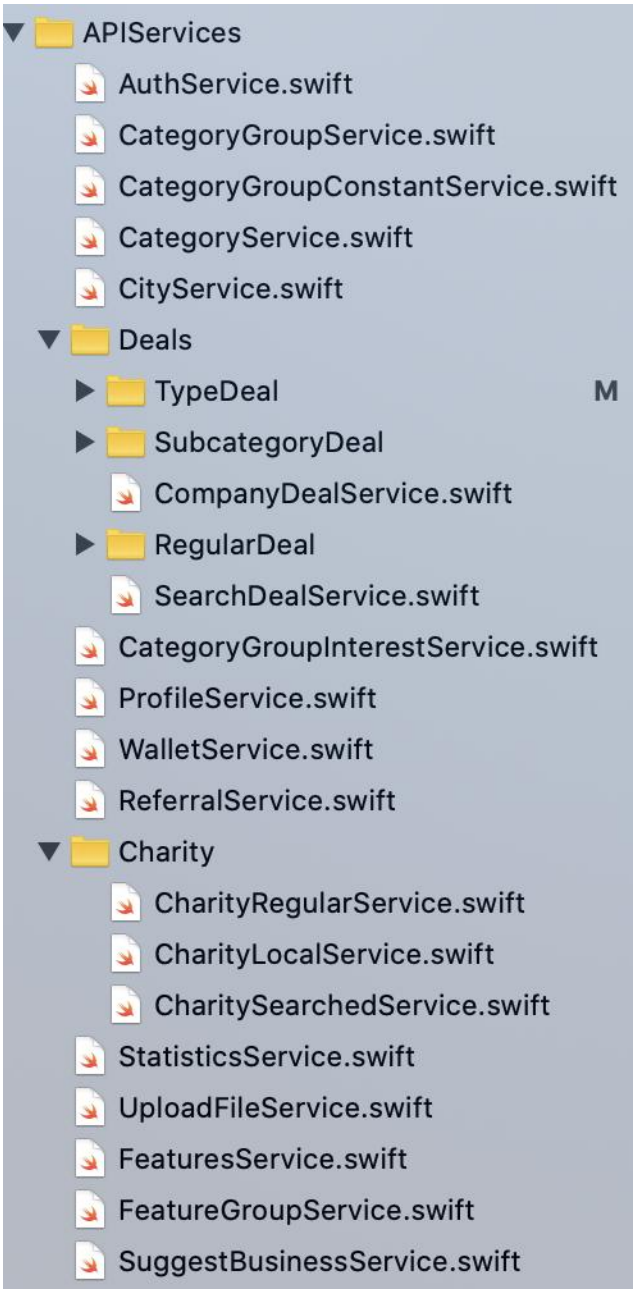


Рисунок 3.4 - Список файлів

Але, насправді, ці файли не містять в собі класів, а тільки розширення до класу моделі, так як Модель є головна за командування маніпуляцій даними. Іншими словами: саме через модель приходять запити від Mediator для отримання або створення даних, тому Модель є відправною точкою подальших запитів.

Для прикладу того, як виконуються запити до сервісу та локальної бази нижче наведений скріншот (рисунок 3.5) з рошириенням однієї із сутностей: City, де один запит віконується для того, щоб завантажити список міст, другий для того, щоб виконувати пошук міст, а третій для того, щоб діставати місто по id із бази. Функція **Socket.shared.send()** - це є використання сервісу, який виконує запит до бекенду, а функція **parse()** структурізує отримані дані та зберігає їх у базу.

Рисунок 3.5 - Розширення сутності City

```
extension City {

    static func getCityList(skip: Int,
                            take: Int,
                            completion: @escaping (Result<Int>) -> Void) {

        Socket.shared.send(with: ReferencesEndPoint.getCityList(skip: skip, take: take)) { (result) in
            guard let data = result.value as? [AnyHashable: Any] else {
                completion(.failure(ServerError.noDataInResponse))
                return
            }

            self.parse(from: data, deleteObjects: skip == 0, completion: completion)
        }
    }

    static func searchCities(by query: String,
                             skip: Int,
                             take: Int,
                             completion: @escaping (Result<Int>) -> Void) {

        Socket.shared.send(with: ReferencesEndPoint.searchCities(query: query, skip: skip, take: take)) { (result) in
            guard let data = result.value as? [AnyHashable: Any] else {
                completion(.failure(ServerError.noDataInResponse))
                return
            }

            self.parse(from: data, deleteObjects: skip == 0, completion: completion)
        }
    }

    static func fetchCity(by id: Int64, in context: NSManagedObjectContext) -> City? {
        return City.mr_findFirst(with: NSPredicate(format: "id == %d", id), in: context)
    }

}
```

Отже, головна ідея щодо того, що Модель вміє себе зберігати, завантажувати та відправляти куди треба і все це завдяки компоненті Service

(сервіс) дотримана. А також основною функцією сервісу є забезпечення спілкування системи із зовнішніми сервісами (backend).

3.3 Реалізація базових класів для View і Mediator

Реалізація модулів View та Mediator є найголовнішою серед усіх інших, тому саме цим двом пунктам (3-4) буде приділено найбільше уваги.

Перед тим, як перейти до реалізації цього пунку, необхідно буде пояснити чому існують базові класи, а чому є реалізації і навіщо взагалі навіщо такий розподіл.

Уявімо, що є так названий констуктруктор, який складається з двох типів елементів: View і Mediator. У кожного типу є декілька видів елементів, але будь-які елементи з кожного типу можуть поєднюватись тільки з елементами іншого типу, таким чином кожна пара представляє собою модуль UI. На рисунку 3.6 зображені групи можливих видів кожного типу: View і Mediator та новоутровена пара, яка складається с одного елементу групи View та одного елементу групи Mediator і являє собою модуль.

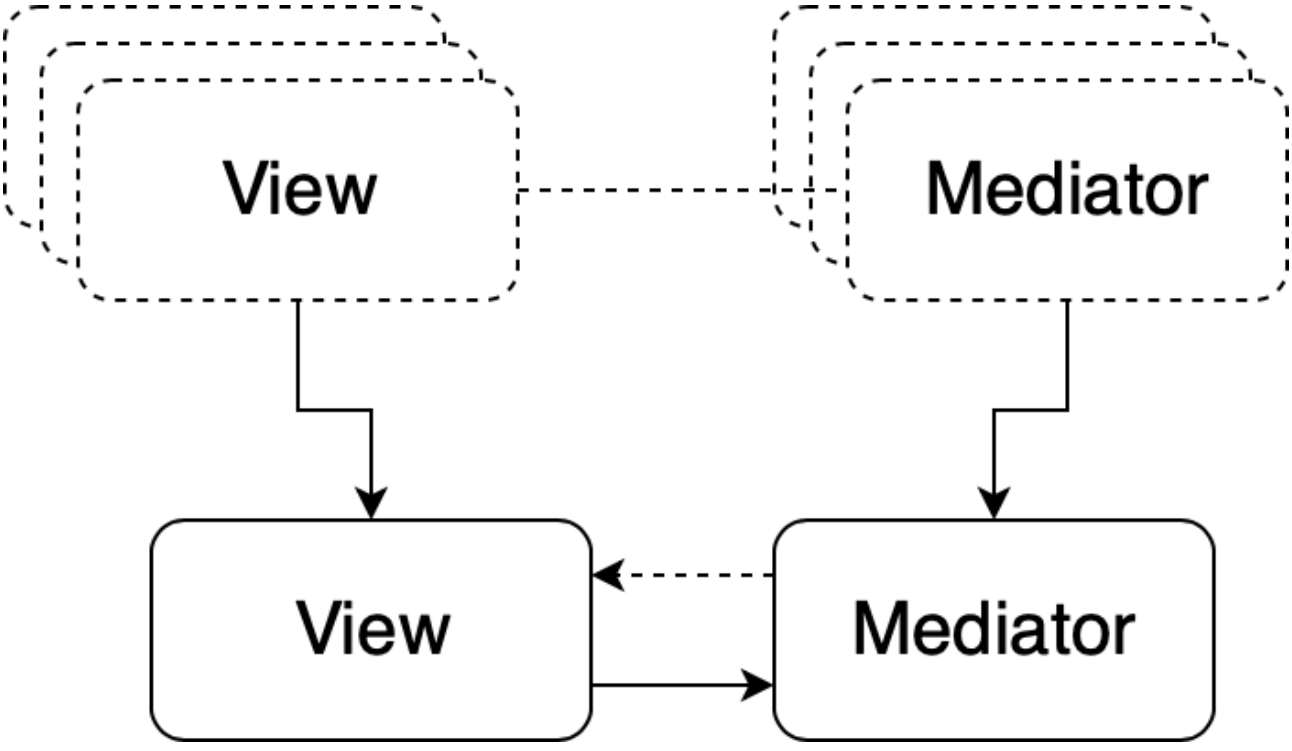


Рисунок 3.6 - 2 групи об’єктів

Наприклад, в будь-якому мобільному застосунку є Таблиці. Таблиці зазвичай завантажують дані та відображають їх, також таблиці мають пагінацію, таблиці мають пошук, фільтрацію, або ж таблиці можуть оновлюватись. Найцікавіше в цьому те, що всі перераховані функціональності можуть зустрічатись як окремо, так і може бути, що всі ці функції будуть в одній таблиці. Забезпечивши такий конструктор и створивши зазделегіть такі елементи, як таблиця з оновленням, таблиця з пагінацією, таблиця з пошуком, можна зекономити багато часу, так як саме реалізація цього конкструктора є головною особливістю архітектури, яка забезпечую перевикористованість. Всі елементи повинні бути написані з самого почтаку розробки та повинні бути перевикористані в екранах застосунку. Цей констуктор з технічного аспекту реалізовується за допомогою наслідуванню і створенню ієрархії класу. Тобто спочатку реалізовуються основні необхідні базові класи (база констуктора), які будуть використовуватись у майбутньому, а далі вже базові класи будуть використовуватись для створення екранів. При подальшій підтримці застосунку може виникнути необхідність додавання нового функіоналу. В такому випадку при наявності необхідних базових класів новий функціонал додається за мінімальний період часу, а якщо необхідного базового класу немає, то створюється відповідний і, вірогідно, вже у майбутньому цей новостворений базовий клас також буде перевикористовуватись.

Тобто головна задача при створенні такого конструктора є економія часу, яка реалізовується за допомогою перевикористання вже написаного коду.

Для більшого розуміння наведемо приклад реалізації конструктора в iOS. Більшість застосунків на iOS використовуються в користувацькому інтерфейсі базову компоненту - Table View (таблиця). Її можна зустріти всюди: в будь-якому месенджері, в застосунку Пошта, Facebook, Instagram, Skype, Slack та в багатьох інших. Тому в якості базових класів буде будуватись ієрархія якраз для цієї компоненти. В даній архітектурі вона розглядається в якості модуля View: сама таблиця як View, а відповідний до нього контролер (TableViewController) як View Controller. Отже, для початку створюється базові

класи для зв’язки View-Mediator - базова таблиця може просто відображати себе з даними. Також повинен існувати базовий Mediator, який буде як раз надавати ці дані.[12]

GenericTableViewController та GenericMediator - ці класи, зв’язані виключно абстракціями. Їхнє спілкування відбувається завдяки шаблону проектування “Делегування”. В iOS цей шаблон реалізовується за допомогою протоколів, колбеків, KVO (key value observing) та Notification Centre (рисунки 3.7-3.8).

```
import Foundation
import CoreData

internal typealias LoadingCallback = () -> Void

internal protocol GenericMediatorProtocol {

    var delegate: GenericMediatorDelegate? { get set }

    var numberOfSections: Int { get }
    var fetchLimit: Int? { get }
    var predicate: NSPredicate? { get }

    func numberOfItems(in section: Int) -> Int
    func item(for indexPath: IndexPath) -> ManagedObject
    func update()

}
```

Рисунок 3.7 - Протокол Mediator для спілкування

```
import Foundation
import CoreData

internal protocol GenericMediatorDelegate: class {
    func willChangeContent()
    func didChangeContent()
    func didChange(object: Any, at indexPath: IndexPath?, newIndexPath: IndexPath?,
       for type: NSFetchedResultsChangeType)
    func didChange(section info: NSFetchedResultsSectionInfo, at sectionIndex: Int,
       for type: NSFetchedResultsChangeType)
}
```

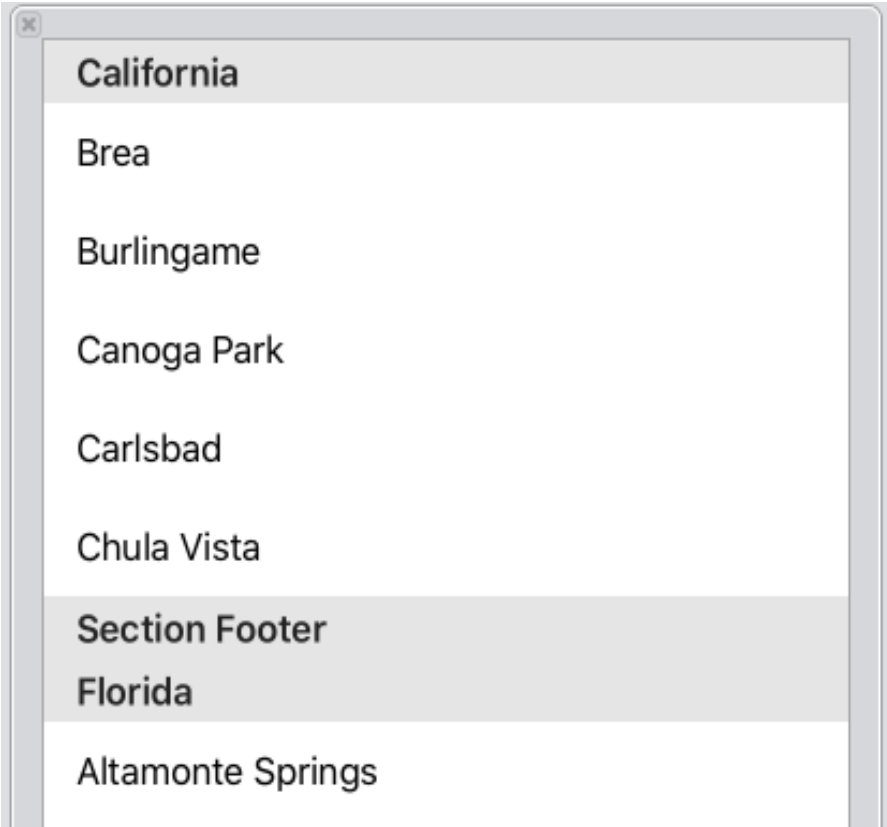
Рисунок 3.8 - Протокол делегату для спілкування

Протокол GenericMediatorProtocol - це протокол для звертання View до Mediator, який містить необхідні методи для відображення numberOfSections (кількість секцій), numberOfItems (кількість рядків), item(for indexPath) (дані для конкретного рядка), тощо.

Протокол GenericMediatorDelegate - це протокол для звертання Mediator до View, який містить необхідні методи, щоб оновляти View. Після змін в даних Mediator за допомогою, наприклад, методу didChangeContent() повідомляє View, що дані змінились, тому він повинен оновитись.

Тобто, будь-який клас, який реалізує протокол GenericMediatorProtocol зможе бути Mediator'ом для GenericTableViewController. Так само і навпаки, будь-який клас, який реалізує методи протоколу GenericMediatorDelegate зможе бути View для Mediator'а. Завдяки цьому досягається рівень абстракції для спілкування, або іншим словом - Розподіленість.

Модуль Model являє собою користувацький інтерфейс у вигляді файлу xib (рисунок 3.9) та відповідний View Controller, який називається GenericTableViewController (рисунок 3.10).



```
internal typealias UITableViewContainer = UITableViewDataSource & UITableViewDelegate

internal class GenericTableViewController
    <Item: ManagedObject>:
UIViewController,
UITableViewContainer,
EmptyStateContainer,
GenericViewModelDelegate,
EmptyStateViewDelegate {

    // MARK: IBs

    @IBOutlet private(set) var tableView: UITableView!

    // MARK: Public properties

    var mediator: GenericMediatorProtocol
    var emptyStateType: DataEmptySetType {
        return .empty
    }

    // MARK: GenericViewModelDelegate

    func willChangeContent() {
        tableView.beginUpdates()
    }

    func didChangeContent() {
        tableView.endUpdates()
        tableView.isUserInteractionEnabled = true
    }

    func didChange(object: Any, at indexPath: IndexPath?, newIndexPath: IndexPath?, for type:
       NSFetchedResultsChangeType) {
        switch type {
        case .insert:
            guard

                let currentIndexPath = newIndexPath else {
                    return
            }
            tableView.insertRows(at: [currentIndexPath], with: .none)
        case .delete:
```

Рисунок 3.9 - Файл xib

Рисунок 3.10 - Клас GenericTableViewController

Користувацький інтерфейс в модулі View може бути не тільки відображений за допомогою файлів xib, а й за допомогою storyboard та навіть за допомогою коду. Apple представила концепцію "Storyboard" в SDK iOS5 для

використання та покращення управління екранами у застосунках. Але все ще можна використовувати можливість розробки через .xib.

Перед Storyboard, кожен UIViewController мав асоційований з ним .xib. Але ж у Storyboard є наступні особливості:

1. .storyboard - це фактично один файл для всіх екранів у застосунку, і він показує поток екранів. Це відбувається завдяки можливості додавання segue (переходи) між екранами. Таким чином, це зводить до мінімуму стандартний код, необхідний для управління декількома екранами.

2. Мінімізує загальне число файлів користувацького інтерфейсу у застосунку.

Але, використання Storyboard призводить до проблем при спробах досягти максимальної кількості можливих до перевикористання компонент. Отже, для максимальної ефективності Нової Архітектури все ж таки радиться використовувати xib файли у модулях View.

Розглянемо створену ієрархію класів на рисунку 3.11.

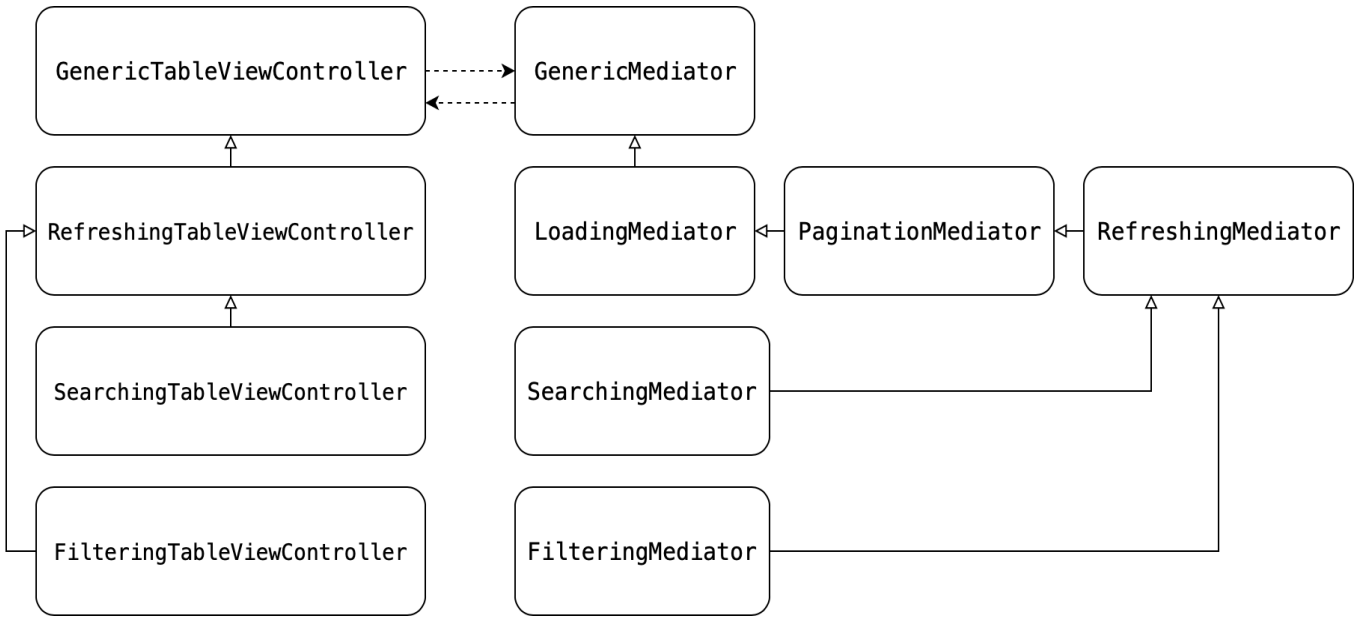


Рисунок 3.11 - Ієрархія класів

Від базового класу GenericTableViewController наслідується RefreshingTableViewController - це таблиця, яка містить усі можливості свого попередника, а також ще може оновлятись завдяки компоненті Pull to Refresh

(свайп зверху вниз). Написавши буквально декілька строк коду (рис. 3.12) та лічені хвилини ми створили нову таблицю, яка вже вміє відображати дані та робити Pull to Refresh. Така таблиця зустрічається в кожному застосунку.

```
internal class RefreshingTableViewController<Item: ManagedObject>: GenericTableViewController<Item> {

    private(set) lazy var refreshControl: UIRefreshControl = {
        let refreshControl = UIRefreshControl()
        refreshControl.addTarget(self,
                                 action: #selector(handleRefresh(_:)),
                                 for: .valueChanged)
        refreshControl.tintColor = .white
        return refreshControl
    }()

    override func viewDidLoad() {
        super.viewDidLoad()

        if
            let mediator = Mediator as? RefreshingMediatorProtocol,
            mediator.refreshIsEnabled {
            tableView.refreshControl = refreshControl
        }
    }

    @objc
    func handleRefresh(_ refreshControl: UIRefreshControl) {
        (mediator as? RefreshingMediatorProtocol)?.refreshData {
            DispatchQueue.asyncInMain {
                self.refreshControl.endRefreshing()
            }
        }
    }

}
```

Рисунок 3.12 - Реалізація RefreshingTableViewController

Щодо відповідного для цього View Controller протоколу Mediator - для створення самого медіатору пропонується поділити його фукнціональні спроможності на декілька різних функціональних Медіаторів: LoadingMediator, PaginationMediator, RefreshingMediator.

LoadingMediatorProtocol наслідується від GenericMediatorProtocol. Цей медіатор додає до можливостей попередника наступні дії: він вміє завантажувати дані та очищати дані (рисунок 3.13).

```
internal protocol LoadingMediatorProtocol: GenericMediatorProtocol {

    func loadObjects(completion: LoadingCallback?)
    func clearObjects(completion: LoadingCallback?)

}
```

Рисунок 3.13 - Протокол LoadingMediatorProtocol

PaginationMediatorProtocol наслідується від LoadingMediatorProtocol та містить в собі логіку, яка відповідає за реалізацію пагінації в таблиці (рис. 3.14). Пагінація дозволяє завантажувати не всі дані за один раз (даних може бути дуже багато і дані будуть завантажуватись дуже довго), а частинам - наприклад, по 15 об’єктів.

Пагінація - під таким терміном мається на увазі контрольний блок, за допомогою якого на сторінці виводиться частина інформації з великого масиву однотипних даних. Інші його назви - pagination, пейджинг, лістинг. В сучасних умовах пагінація придбала форму, якої зручно користуватися.

Існує безліч методів реалізації посторінкового навігації з різним функціоналом. Найбільш поширеними з них вважаються:

- Пряма (універсальна) пагінація з порядковою нумерацією (1-2-3). Це перелік номерів страніц.Чаще всього також додають кнопки "далі / назад" по обидва боки посилань на сторінки.
- Діапазон вибору із зазначенням позиції в лістингу (1-10, 20-59).
- Зворотного типу ("нове", 100-21, 20-1).

```
internal protocol PaginationMediatorProtocol: LoadingMediatorProtocol {

    var takeLimit: Int { get }
    var totalPages: Int { get set }
    var currentPage: Int { get set }

}
```

Рисунок 3.14 - Протокол PaginationMediatorProtocol

І, нарешті, третій протокол RefreshingMediatorProtocol, який наслідується від протоколу PaginationMediator (рис. 3.15). Отже, цей медіатор вже вміє завантажувати дані як цілком, так і частинами. Ціль RefreshingMediator в тому, щоб оновлювати вже існуючі (завантажені) дані, які можуть залежати від якогось зовнішнього фактору (як приклад, від локації).

```
internal class RefreshingMediator<Item: ManagedObject>: PaginationMediator<Item>,
RefreshingMediatorProtocol, LocationMediatorProtocol {

    var currentLocation: LocalLocation? = LocalLocation()

    var refreshIsEnabled: Bool {
        return true
    }

    func refreshData(completion: @escaping LoadingCallback) {
        currentLocation = LocalLocation()
        currentPage = 0
        loadObjects(completion: completion)
    }

}
```

Рисунок 3.15 - Протокол RefreshingMediatorProtocol

Отже, контролеру RefreshingTableViewController відповідає медіатор RefreshingMediator, який був створений завдяки трьом своїм попередникам: GenericMediator, LoadingMediator, PaginationMediator.

Від RefreshingTableViewController наслідується SearchingTableViewController - це таблиця, яка має ключову особливість: вона містить компоненту для пошуку в таблиці (рис. 3.16). Це одна із найпопулярніших таблиць, яка використовується майже в усіх застосунках. Для забезпечення цієї функціональності до вже існуючої таблиці Refreshing (яка вміє оновлюватись) додається проста компонента UI - Search Bar (рис. 3.17). Search Bar – це основна компонента в iOS, яка відповідає за процес пошуку по таблиці, до якої вона відноситься. При введенні нових символів у поле Search Bar'у відбуваються запити на оновлення таблиці. Тобто навички оновлення від попередника використовуються при запитах на оновлення при введенні даних у пошуковий рядок.

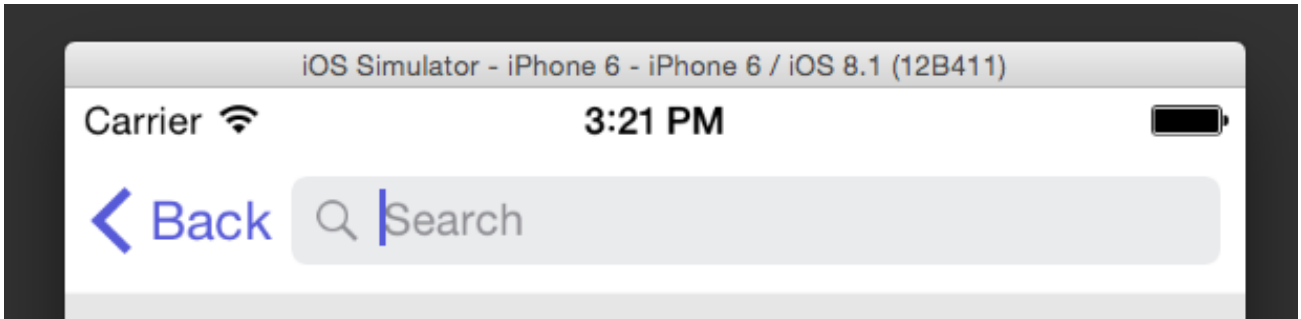


Рисунок 3.17 - Search Bar

Рисунок 3.16 - Клас SearchingTableViewController

```
internal class SearchingTableViewController<Item: ManagedObject>:
   RefreshingTableViewController<Item>,
    UISearchResultsUpdating,
    UISearchBarDelegate {

    let searchController = UISearchController(searchResultsController: nil)

    override func viewDidLoad() {
        super.viewDidLoad()

        setupSearch()
    }

    // MARK: - UISearchResultsUpdating
    func updateSearchResults(for searchController: UISearchController) {
        if let mediator = mediator as? SearchingMediator<Item> {
            mediator.searchQuery = searchController.searchBar.text
        }
    }

    // MARK: - UISearchBarDelegate
    func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
        searchBar.resignFirstResponder()
        (mediator as? RefreshingMediatorProtocol)?.refreshData {}
    }

    func searchBarCancelButtonClicked(_ searchBar: UISearchBar) {
        searchBar.resignFirstResponder()
        if let mediator = mediator as? SearchingMediator<Item> {
            mediator.searchQuery = .none
        }
    }
```

Для підтримки функціональності цього Search Bar'у у SearchingTableViewController створено відповідний медіатор - SearchingMediator (рис. 3.17).

```
internal protocol SearchingMediatorProtocol: PaginationMediatorProtocol {

    var searchQuery: String? { get set }

}
```

Рисунок 3.17 - Протокол SearchingMediatorProtocol

Медіатор SearchingMediator наслідується від PaginationMediator, так як нам необхідні функції завантаження цілком, очищення та пагінація для реалізації функціональності пошуку.

Також у RefreshingTableViewController є ще один спадок FilteringTableViewController - таблиця, яка має функціональність фільтрування даних (рис. 3.18). Тобто користувач зможе фільтрувати таблицю так, щоб відображались тільки ті дані, які йому потрібні.

```
internal class FilteringTableViewController<Item: ManagedObject>:
RefreshingTableViewController<Item>, Filterable, DataUpdatable {

    override func viewDidLoad() {
        super.viewDidLoad()

        if (mediator as? FilteringMediatorProtocol) != nil {
            addRightBarButtonItem(title: LocalizedStrings.filter, action:
               #selector(filterAction))
            setupFilter()
        }
    }

    @objc
    func filterAction() {
        guard let filter = (mediator as? FilteringMediatorProtocol)?.filter else {
            return
        }

        present(FilterNavigationController(filter: filter, delegate: self), animated: true,
           completion: nil)
    }

    func getFilter() -> Filter {
        return Filter.shared
    }

    // MARK: DataUpdatable

    func updateData() {
        (mediator as? RefreshingMediatorProtocol)?.refreshData {
            self.mediator.update()
            self.tableView.reloadData()
        }
    }
```

Рисунок 3.18 - Таблиця FilteringTableViewController

Як видно в реалізації FilteringTableViewController використовуються різні запити до медіатору, так як медіатор FilteringMediatorProtocol наслідується від PaginationMediatorProtocol. Тому при необхідності таблиця після застосування фільтру звертається до медіатору із запитом на оновлення даних,

а також передає налаштування фільтрації до медіатору. Таким чином FilteringMediatorProtocol містить в собі логіку по роботі з фільтруванням (рис. 3.19).

```
internal protocol FilteringMediatorProtocol: PaginationMediatorProtocol {

    var filter: Filter? { get set }

}
```

Рисунок 3.19 - Протокол FilteringMediatorProtocol

Всі базові класи можуть бути об’єднані в різні комбінації. Наприклад, може бути створений еркан, в якому будуть необхідні наступні функціональності: таблиця, яка може оновлюватись, яка містить пагінацію, а також пошук і фільтрація. Ми маємо вже готові два екрани FilteringTableViewController і SearchingTableViewController. Для цього створюється новий клас FilterSearchingCollectionViewController за допомогою наслідування від SearchingTableViewController та реалізації протоколу Filterable (рис. 3.20). Також створюється необхідний медіатор FilterSearchingMediator, який просто наслідується від SearchingMediator та реалізовує протокол для фільтрації - FilteringMediatorProtocol.

```
internal class FilterSearchingMediator<Item: ManagedObject>:
    SearchingMediator<Item>,
    FilteringMediatorProtocol,
    CLLocationManagerDelegate {

    var filter: Filter? {
        didSet {
            currentPage = 0
        }
    }

    // MARK: Initialization
    override init() {
        super.init()
        LocationManager.subscribe(self)
    }
```

Рисунок 3.21 - Медіатор FilterSearchingMediator

```
internal class FilterSearchingCollectionViewController<Item: ManagedObject>:
   SearchingCollectionViewController<Item>,
Filterable,
DataUpdatable {

    override func viewDidLoad() {
        super.viewDidLoad()

        if (mediator as? FilteringMediatorProtocol) != nil {
            addRightBarButtonItem(title: LocalizedStrings.filter, action:
               #selector(filterAction))
            setupFilter()
        }
    }

    func getFilter() -> Filter {
        return Filter.shared
    }

    func setupFilter(_ filter: Filter?) {
        let newFilter = filter ?? getFilter()
        (mediator as? FilterSearchingMediator<Item>)?.filter = newFilter

        if let filter = filter {
            Filter.shared.saveFilter(filter)
        }

        if
            let filter = (mediator as? FilteringMediatorProtocol)?.filter {
            var postfix = ""
            if let count = filter.filtersAmount {
                postfix = " (\(String(describing: count)))"
            }
            navigationItem.rightBarButtonItem?.title = LocalizedStrings.filter + postfi
        }
    }
```

Рисунок 3.20 - Клас FilterSearchingCollectionViewController

Отже, на створення нового базового екрану знадобилось зовсім не багато часу, а найголовніше, що кількість строк коду в кожному класу не перевищує 50.

Одна із головних ідей цього конструктора також є те, що кожний базовий клас може бути використаний для будь-якої сутності моделі. Тобто, будь-яка таблиця може відображати список об’єктів будь-якої сутності. Це реалізовано за допомогою механізму Generics.

У світі мов програмування існує парадигма узагальненої розробки. Вона полягає в можливості створювати алгоритми, однаково працюють з різними, заздалегідь невідомими типами вхідних даних.

Для її реалізації Swift володіє дженериками, які вже в повній мірі задіяні в стандартній вбудованій бібліотеці. Наприклад, масиви і словники, де тип елементів невідомий аж до ініціалізації, побудовані з ними.

Можна сказати, що методи, структури і класи, написані із застосуванням дженериків, здатні приймати типи в якості параметрів і використовувати їх в коді.

За допомогою мови програмування безпечного по відношенню до типів, це звичайна проблема як написання коду, який діє тільки на один тип, але цілком коректний і для іншого типу. Уявіть собі, наприклад, що функція додає два цілих числа. Функція, яка додає два числа з плаваючою комою, виглядала б дуже схожою - але фактично, вона буде виглядати ідентично.

Єдиною відмінністю буде тип значення змінних.

У мові із суворим контролем типів Ви повинні були б визначити окремі функції як addInts, addFloats, addDoubles, і т.д., де у кожної функції був правильний аргумент, і типи значень, що повертаються.

Багато мов програмування здійснюють вирішення цієї проблеми. C ++, наприклад, використовує шаблони. Swift, як Java і C # використовують Generics (дженерики).

Так як таблиці працюють з будь-якими даними однаково, то було запропоновано уникнути дублювання коду за допомогою узагальнення класів Mediator та ViewController.

Отже, всі базові класи обов’язково повинні бути Дженеріками, інакше особливість Нової Архітектури про перевикористання коду стане неправдивою. Реалізація Дженеріків позначається в кутових дужках (рис. 3.21).

```
internal class RefreshingCollectionViewController<Item: ManagedObject>:
   GenericCollectionViewController<Item> {
```

Рисунок 3.21 - Реалізація Generics

Item - це узагальнена назва об’єктів моделі, які реалізовують протокол ManagedObject.

Наведений список базових класів є виключно прикладом і, звичайно ж, не є обов’язковим для реалізації в кожному проекті, він може змінюватись згідно з вимогами застосунку. Але для реалізації обраного застосунку ZipHub підходять

саме такі базові класи, так як переважна частина екранів містить саме ці розглянуті функції.

3.4 Реалізація конкретних екранів (конкретних View і Mediator)

Розглянемо реалізацію наступних екранів для застосунку ZipHub:

1. Головна сторінка.
2. Список в обраній категорії.
3. Пошук.
4. Гаманець.

1. Головна сторінка

Головна сторінка в застосунку ZipHub називається Shop (рис. 3.22).

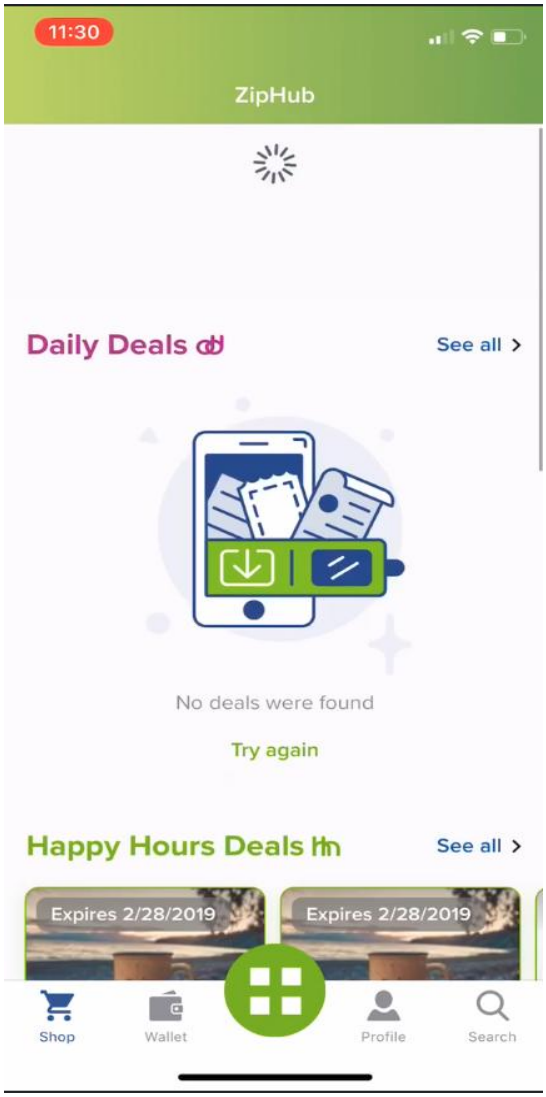


Рисунок 3.22 - Головний екран Shop

Екран Shop це таблиця, яка в кожному рядку містить ще одну горизонтальну таблицю. Shop містить в собі функції завантаження та оновлення даних. Значить, для реалізації будемо використовувати базовий клас RefreshingTableViewController (рис.3.23).

```
internal class ShopViewController: RefreshingTableViewController<DealsCategory>,
  CompanyPresentable,
  DealPresentable,
  Suggestable {

  // MARK: IBs

    @IBOutlet private weak var tableView: UITableView!

    @IBOutlet private weak var loadingView: UIView!

    var isLoading = true {
        didSet {
            self.loadingView.isHidden = true
            if !Defaults[.isNotInitialShopLaunch] {
                proceedInTutorial()
                Defaults[.isNotInitialShopLaunch] = true
            }
        }
    }

    // MARK: Private properties

    private let mediator: ShopMediatorProtocol

    private lazy var refreshControl: UIRefreshControl = {
        let refreshControl = UIRefreshControl()
        refreshControl.addTarget(self,
                                 action: #selector(handleRefresh(_:)),
                                 for: .valueChanged)
        return refreshControl
    }()

    private var requestWorkItem: DispatchWorkItem!

    private var cells = Set<UITableViewCell>()
```

Рисунок 3.23 - Реалізація ShopViewController

ShopViewController наслідується від RefreshingTableViewController і вже містить в собі логіку по завантаженню та оновленню даних (по pull to refresh), а також завдяки Generics ми вказуємо назву сутності моделі - DealsCategory.

Для реалізації відповідного медіатору ShopMediator (рис. 3.24) будемо використовувати готовий базовий клас RefreshingMediator, знову ж таки, з сутностю DealsCategory, бо відображаємо в таблиці список об’єктів DealsCategory.

```
internal class ShopMediator: RefreshingMediator<DealsCategory> {

    // MARK: Public properties

    var items: [ShopItem] = [.daily, .happyHours, .nearby]

    lazy var loadStatuses: [Bool] = {
        var array = Array(repeating: false, count: numberOfSections)
        return array
    }()

    var numberOfSections: Int {
        return items.count
    }

    weak var delegate: ShopViewModelDelegate?

    // MARK: Initialization

    override init() {
        super.init()
        LocationManager.subscribe(self)
    }

    // MARK: Public methods

    func loadObjects(completion: @escaping (Bool) -> Void) {
        DispatchQueue.global(qos: .userInitiated).async {
            let dispatchGroup = DispatchGroup()

            for item in ShopItem.allCases {
                dispatchGroup.enter()
                item.loadObjects { _ in
                    dispatchGroup.leave()
                }
            }

            let status = dispatchGroup.wait(timeout: .now() + Socket.timeoutValue)

            DispatchQueue.asyncInMain {
                completion(status == .success)
            }
        }
    }

    func loadObjects(for item: ShopItem, completion: @escaping (Result<Int>) -> Void) {
        item.loadObjects(completion: completion)
    }

}
```

Рисунок 3.24 - Реалізація ShopMediator

2. Список в обраній категорії

Список в обраній категорії - це таблиця, яка має функції оновлення, пагінації та фільтрації (рис. 3.25).

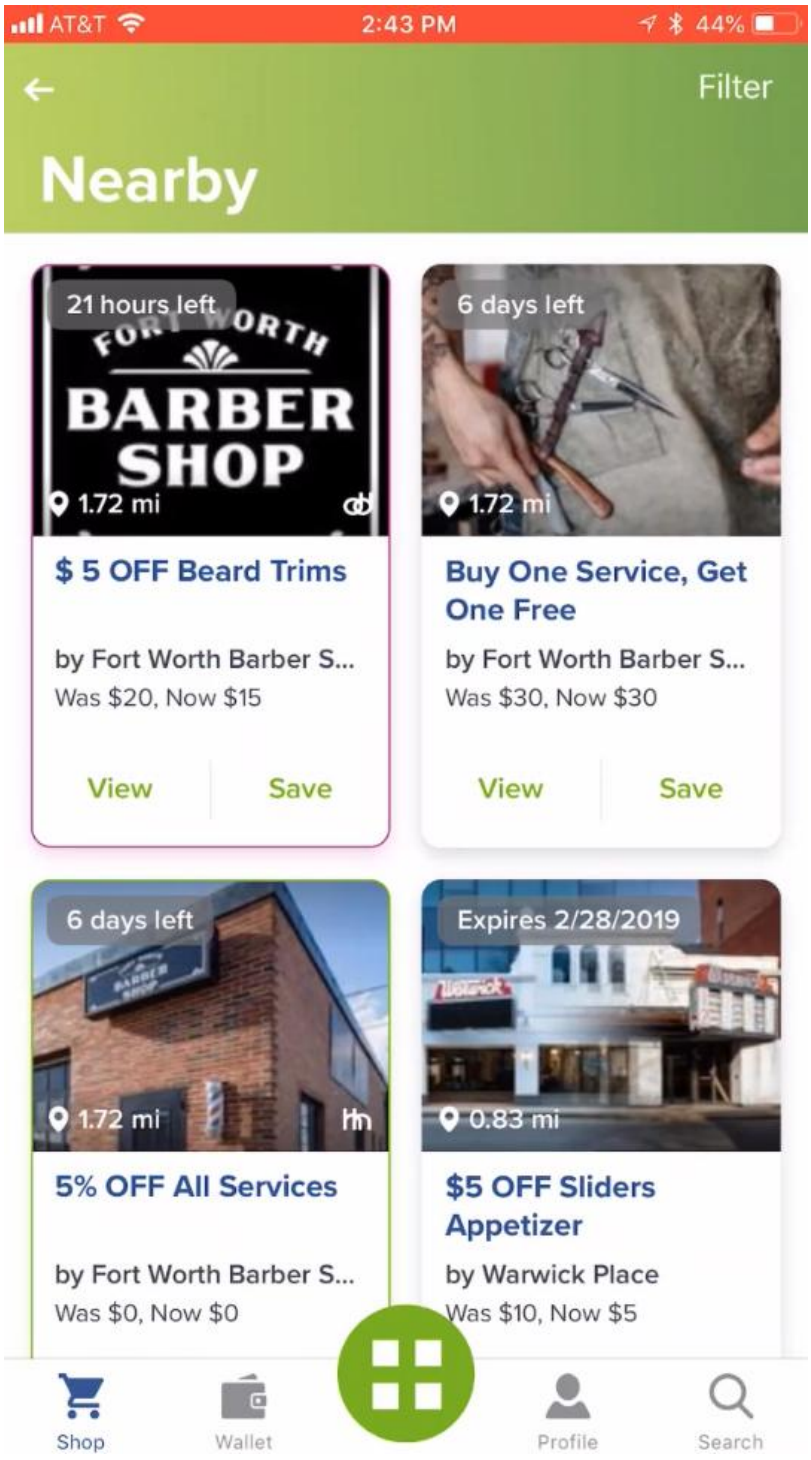


Рисунок 3.25 - Екран Список в обраній категорії

Для реалізації даного екрану достатньо буде використати вже існуючий готовий базовий клас, який називається FilteringCollectionViewController. Новий клас SpecificTypeDealsViewController (рис. 3.26) наслідується від FilteringCollectionViewController та встановлює для відображення об'єктів сутність FilteredTypeDeal.

```
internal class SpecificTypeDealsViewController:
   FilteringCollectionViewController<FilteredTypeDeal>,
    CompanyPresentable,
    DealPresentable,
    Suggestable {

    override var emptyStateType: DataEmptySetType {
        return .deals
    }

    override func viewDidLoad() {
        super.viewDidLoad()

        collectionView.register(cellType: DealCollectionViewCell.self)

        navigationController?.navigationBar.prefersLargeTitles = true
        if let mediator = mediator as? SpecificTypeDealsMediator,
            let name = mediator.type.dealType?.fullName {
            title = name
        }

        (mediator as? LoadingMediatorProtocol)?.loadObjects {}
    }

    override func viewWillAppear(_ animated: Bool) {
        super.viewWillAppear(animated)
        navigationController?.navigationBar.prefersLargeTitles = true
    }

    override func getFilter() -> Filter {
        var filter = super.getFilter()
        guard let model = mediator as? SpecificTypeDealsMediator else {
            return filter
        }

        guard let path = model.type.dealType?.path else {
            return filter
        }
        filter.dealTypes = [path]
        filter.dealTypeBlocked = true

        return filter
    }
```

Рисунок 3.26 - Клас SpecificTypeDealsViewController

Для створення медіатору SpecificTypeDealsMediator (рис. 3.27) для цього контролеру візьмемо за основу клас FilteringMediator, який вже реалізував основну важку логіку по виконанню фільтрації, запитів на завантаження даних та пагінацію. Все, що треба зробити це наслідуватись від цього класу та перевизначити деякі необхідні методи.

```
internal class SpecificTypeDealsMediator: FilteringMediator<FilteredTypeDeal> {

    // MARK: Private properties

    private(set) var type: DealSource

    // MARK: FilteringMediator

    override var predicate: NSPredicate? {
        get {
            var predicate: NSPredicate?
            if let path = type.dealType?.path {
                predicate = NSPredicate(format: "dealType == %@", path)
            }
            return predicate
        }
        set {}
    }

    override func loadObjects(completion: LoadingCallback?) {
        if let path = type.dealType?.path {
            FilteredTypeDeal.getFilteredSpecificDeals(type: path,
                                                      skip: skipAmount,
                                                      take: takeLimit,
                                                      location: currentLocation,
                                                      filter: filter,
                                                      completion: completion)
        } else if let id = type.categoryId {
            FilteredSubcategoryDeal.getFilteredDealsById(id: id,
                                                         skip: skipAmount,
                                                         take: takeLimit,
                                                         location: currentLocation,
                                                         filter: filter,
                                                         completion: completion)
        }
    }

}
```

Рисунок 3.27 - Медіатор SpecificTypeDealsMediator

3. Пошук

Пошук - це такий екран, який містить таблицю з насутпними функціями: оновлення, пагінація, фільтрація та пошук (рис. 3.28).

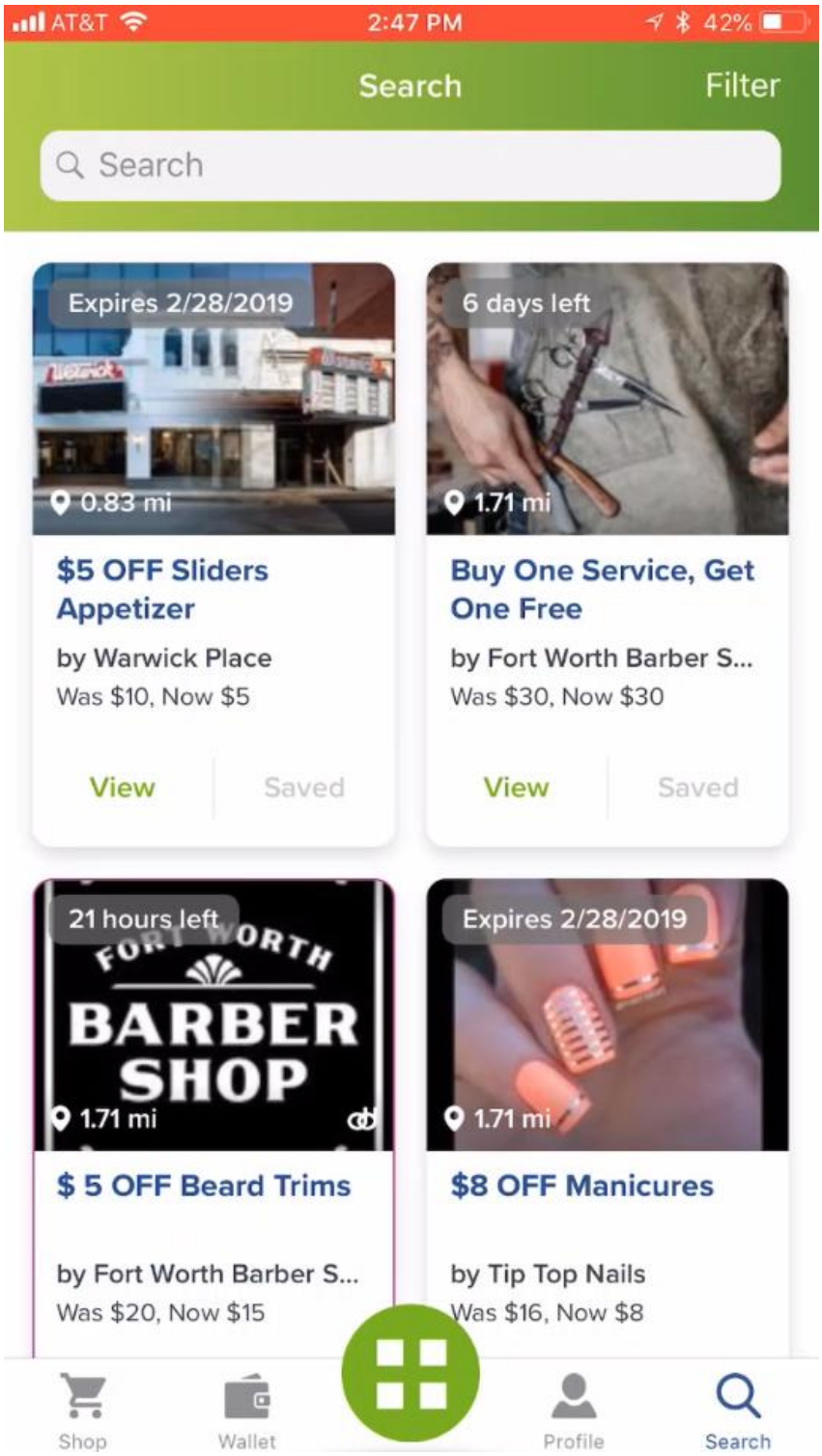


Рисунок 3.28 - Екран Пошук

Для реалізації цього екрану як раз буде використовуватись базовий клас, який був створений останнім - для прикладу того, як можна комбінувати основні базові класи. Для цього створимо клас SearchViewController (рис. 3.29), який наслідується від новоствореного базового класу FilterSearchingCollectionViewController із сутностю SearchDeal.

```
internal class SearchViewController: FilterSearchingCollectionViewController<SearchDeal>,
    CompanyPresentable, DealPresentable, Suggestable {

    // MARK: Public properties

    override var emptyStateType: DataEmptySetType {
        return .deals
    }

    // MARK: Initialization

    override func viewDidLoad() {
        super.viewDidLoad()
        title = "Search"

        collectionView.register(cellType: DealCollectionViewCell.self)
        updateData()
    }

    override func viewWillAppear(_ animated: Bool) {
        super.viewWillAppear(animated)
        navigationController?.makeVisibleNavigationBar()
    }

    override func willMove(toParent parent: UIViewController?) {
        super.willMove(toParent: parent)
        (navigationController as? BasicNavigationController)?.setNavigationBarGradient()
        navigationController?.navigationBar.prefersLargeTitles = false
    }

    deinit {
        NotificationCenter.default.removeObserver(self)
    }

    override func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath: IndexPath) ->
       UICollectionViewCell {
        let cell: DealCollectionViewCell = collectionView.dequeueReusableCell(for: indexPath)
```

Рисунок 3.29 - Клас FilterSearchingCollectionViewController

Вся основна необхідна логіка роботи фільтрації, пошуку, пагінації і оновлення вже працює. Залишається вносити лише маленькі корективи по UI частині (наприклад, зміна назви екрану чи нові взаємодії за таблицею). Для обробки цієї логіки створюємо відповідний медіатор SearchMediator (рис. 3.30),

```
internal class SearchMediator: FilterSearchingMediator<SearchDeal> {

    // MARK: Public properties

    override var refreshIsEnabled: Bool {
        return false
    }

    override func loadObjects(completion: LoadingCallback?) {
        SearchDeal.getSearchDeals(searchWord: searchQuery ?? "",
                                  skip: skipAmount,
                                  take: takeLimit,
                                  location: currentLocation,
                                  filter: filter) { result in
                                    switch result {
                                    case .failure:
                                        completion?()
                                    case .success(let count):
                                        self.totalPages = count
                                        completion?()
                                    }
        }
    }

    override func clearObjects(completion: LoadingCallback?) {
        SearchDeal.clearDeals { (result) in
            if case .success = result {
                self.totalPages = 0
            }
            completion?()
        }
    }

}
```

який наслідується від класу FilterSearchingMediator зі сутностю SearchDeal.

Рисунок 3.30 - Клас SearchMediator

Цей медіатор, який містить найбільше і найважчу логіку містить в собі всього лиш 40 строк коду! Цей екран був реалізований всього лиш за пару годин.

4. Гаманець

Екран Гаманець - це таблиця, яка вміє тільки оновлюватись завдяки Pull to Refresh (рис. 3.31).

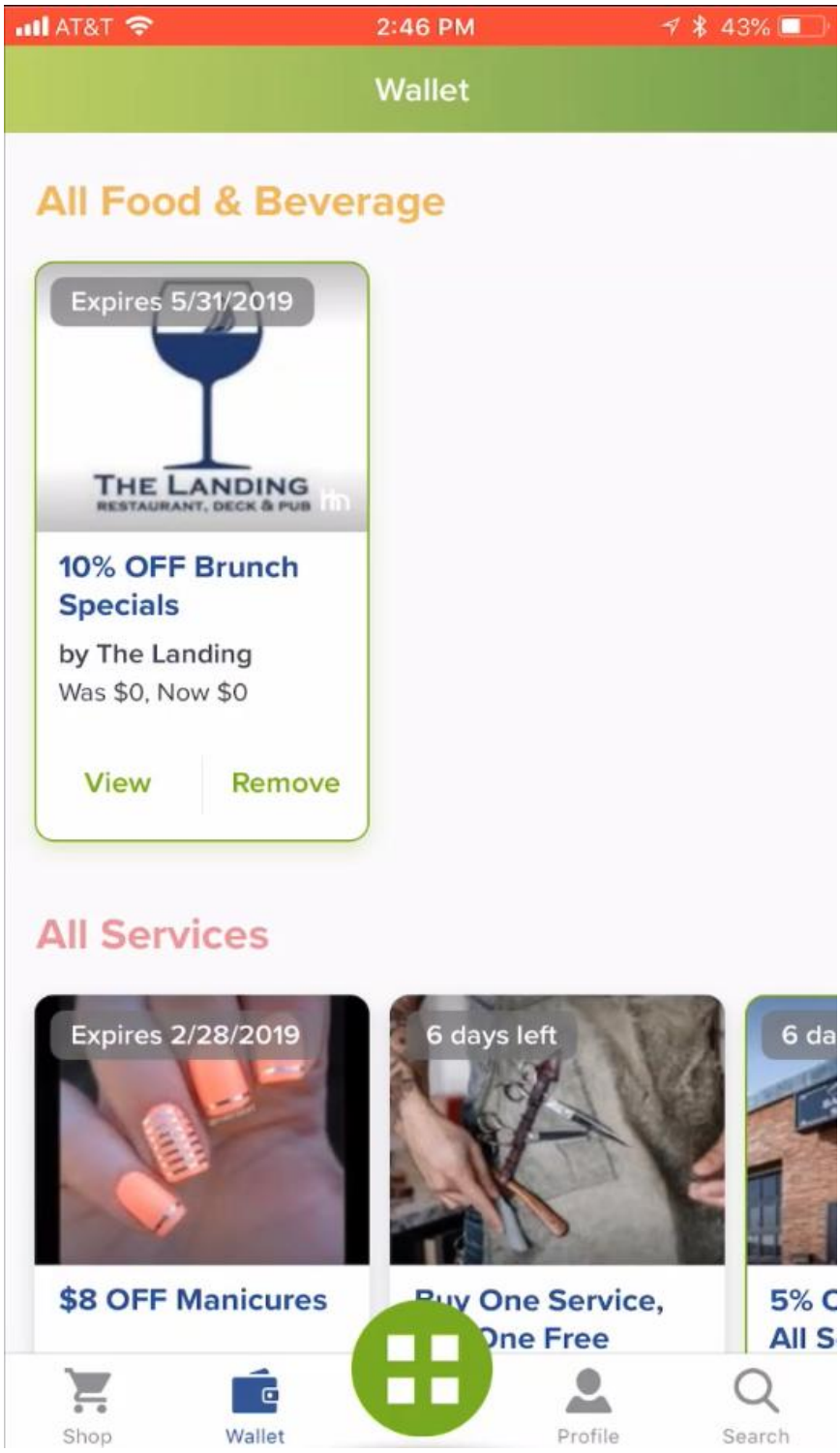


Рисунок 3.31 - Екран Гаманець

Це найпростіший для реалізації екран, так як він містить в собі лише одну функцію - оновлення, і тому для реалізації класу WalletViewController буде використовуватись готовий базовий клас RefreshingTableViewController (рис. 3.32) зі сутністю WalletCategoryGroup.

```
internal class WalletViewController: RefreshingTableViewController<WalletCategoryGroup>, CompanyPresentable
   DealPresentable {

    // MARK: DZNEmptyStateDataSource

    override var emptyStateType: DataEmptySetType {
        return .wallet
    }

    var isTracked: Bool {
        return false
    }

    var companyIsTracked: Bool {
        return false
    }

    override func viewDidLoad() {
        super.viewDidLoad()

        tableView.register(cellType: DealsTableViewCell.self)
        tableView.register(headerFooterViewType: DealsHeaderView.self)

        refreshControl.tintColor = .gray

        title = "Wallet"
    }

    override func viewWillAppear(_ animated: Bool) {
        super.viewWillAppear(animated)

        (mediator as? LoadingMediatorProtocol)?.loadObjects {}
    }

    override func willMove(toParent parent: UIViewController?) {
        (navigationController as? BasicNavigationController)?.setNavigationBarGradient()
        navigationController?.navigationBar.prefersLargeTitles = false
    }
```

Рисунок 3.32 - Реалізація WalletViewController

Для створення необхідного класу медіатора WalletMediator за основу

береться базовий клас RefreshingMediator і встановлюється сутність WalletCategoryGroup (рис. 3.33).

```
internal class WalletMediator: RefreshingMediator<WalletCategoryGroup> {

    var dealsCount: Int = 0

    func removeDealFromWallet(_ deal: WalletCategoryDeal, completion: (() -> Void)?) {
        WalletCategoryDeal.removeDeal(id: deal.id) { _ in
            self.dealsCount -= 1
            completion?()
        }
    }

    override func loadObjects(completion: LoadingCallback?) {
        WalletCategoryGroup.getSavedDeals(location: LocalLocation()) { (result) in
            switch result {
            case let .success(totalPages):
                self.dealsCount = totalPages
                completion?()
            case let .failure(error):

                logger.verbose(
                    """
                    Error has occured during fetchin \(Deal.self) objects.\n
                    ErrorDescription: \(error.localizedDescription)
                    """
                )
                completion?()
            }
        }
    }

}
```

Рисунок 3.3 - Медіатор WalletMediator

Як видно по рисункам з кодом, вся логіка нових екранів уміщається до 50-ти строк коду, а отже і часу для розробки при такій архітектурі буде займати набагато менше.

3.5 Висновки до розділу

Отже, для демонстації застосування на практиці Нової Архітектури було розроблено новий мобільний застосунок для платформи iOS, який називається ZipHub. Було проаналізовано перед початком роботи чому саме Нова Архітектура підходить для розробки даного застосунку, а також після

перегляду дизайну майбутнього застосунку були виділені основні базові класи для модуля Mediator. Завдяки застосуванню Нової Архітектури у застосунку розробка зайняла мінімальну кількість часу, а також для демонстрації було впроваджено нову функціональність. Під час ходу розробки були детально розглянуті запропоновані ефективні підходи, середни були такі, які необхідно застосовувати та такі, що рекомендуються. Для кращого візуального сприйняття Нової Архітектури у кінці розробки було продемонстровано на прикладах готових екранів застосування основних підходів.

4 ПОРІВНЯННЯ З ІСНУЧИМИ АРХІТЕКТУРАМИ

Порівняння є одним із ключових факторів для доведення переваги Нової Архітектури серед інших. Нова Архітектура буде порівнюватись з популярним і дуже відомим архітектурним підходом Model-View-Controller (MVC). Для цього було розроблено таки самий додаток, який використовувався для демонстрації Нової Архітектури, тільки використовуючи архітектуру MVC. Для ефективного проведення порівняння було запропоновано провести наступні етапи тестування Нової Архітектури:

1. Затрачений час для розробки застосунку ZipHub.
2. Порівняння кількості коду в основних класах.
3. Порівняння можливості впровадження нової функціональності.
4. Покриття Unit-тестами.

4.1 Затрачений час для розробки застосунку ZipHub.

Основним та найголовнішим порівнянням є порівняння затраченого часу на розробку повноцінного застосунку. Для прикладу був взятий застосунок ZipHub, який розроблявся двома шляхами: застосовуючи Нову Архітектуру та застосовуючи MVC. Були поставлені задачі для розробки: 1 розробник, 10 екранів та 1 функціональність для впровадження після закінчення розробки. Розробка застосунку планується не для комерційних цілей, тому деталізоване і глибоке тестування не буде враховане в цей час.

Отже, спочатку був засічений час для розробки застосунку з 0 з Новою Архітектурою, а потім засічений час для розробки з MVC. У таблиці 4.1 наведені результати порівняння першого етапу.

В Новій Архітектурі передбачається трата часу на написання «неіснуючих екранів», а тобто базових екранів. Тому такий пункт

підраховується тільки для Нової Архітектури. Для MVC до цього пункту буде вказано, що не було затрачено часу взагалі.

Таблиця 4.1 – Результати першого етапу

	Нова Архітектура	MVC
Час розробки всього застосунку	3,25 місяця	6 місяців
Час розробки базових екранів	3 тижні	0 тижнів
Час розробки 1 екрану	1 тиждень	3 тижні
Час впровадження нової функціональності	0.5 тижня	3 тижні

Згідно з результатами затраченого часу можна впевнено сказати, що Нова Архітектура безумовно економить час та пришвидшує розробку і впровадження нових функціональностей у подальшому.

4.2 Порівняння кількості коду в основних класах

Одним із головних факторів гарної архітектури є кількість написаного коду для впровадження нової функціональності. Для цього в існуючий додаток ZipHub буде додано один новий екран. Після проведення попереднього порівняння вже існують дві реалізації одного застосунку з MVC підходом та Новою архітектурою. Порівняємо кількість рядків коду в двох проектах для створення екрану пошуку, який включає в себе наступні можливості: завантаження даних, оновлення даних, пагінація, фільтрація та пошук даних.

В проекті ZipHub з реалізованою архітектурою Нова архітектура створено клас SearchViewController, який відповідає всім необхідним вимогам (рис. 4.1).

Отже, при кінцевому результаті в класі SearchViewController маємо 89 рядків коду.

```
        return cell
    }

    override func collectionView(_ collectionView: UICollectionView,
                                 layout collectionViewLayout: UICollectionViewLayout,
                                 sizeForItemAt indexPath: IndexPath) -> CGSize {
        let defaultHeight = DealCollectionViewCell.defaultHeight

        let itemsPerRow: CGFloat = 2

        let sectionInsets = UIEdgeInsets(top: 20, left: DealCollectionViewCell.defaultSpacing, bottom: 20, right: 16)

        let paddingSpace = sectionInsets.left * (itemsPerRow + 1)
        let availableWidth = view.frame.width - paddingSpace
        let widthPerItem = availableWidth / itemsPerRow

        return CGSize(width: widthPerItem, height: defaultHeight)
    }

    @objc
    func onDidChangeActiveTabBar(_ notification: Notification) {
        guard (notification.object as? SearchNavigationController) != nil else {
            return
        }
        clearSearch()
    }

    override func collectionView(_ collectionView: UICollectionView,
                                 layout collectionViewLayout: UICollectionViewLayout,
                                 minimumInteritemSpacingForSectionAt section: Int) -> CGFloat {
        return DealCollectionViewCell.defaultSpacing
    }

    override func emptyStateViewDidTapAction(with state: DataEmptySetType?) {
        showSuggestBusiness()
    }

}
```

Рисунок 4.1 - Клас SearchViewController

Для порівняння тепер буде проведена аналогічна розробка екрану Пошук для іншого проекту.

В проекті ZipHub з реалізованою архітектурою MVC створено клас SearchUsersViewController (рис. 4.2), який також відповідає всім необхідним вимогам: завантаження даних, оновлення даних, пагінація, фільтрація та пошук даних.

Отже, при кінцевому результаті в написаному класі SearchUsersViewController маємо 326 рядків коду.

```
291             preconditionFailure("Unexpected segue identifier")
292         }
293     }
294 }
295
296 extension SearchUsersViewController: UISearchResultsUpdating {
297     func updateSearchResults(for searchController: UISearchController) {
298
299     }
300 }
301
302 extension SearchUsersViewController: UISearchBarDelegate {
303
304     func searchBar(_ searchBar: UISearchBar, textDidChange searchText: String) {
305         NSObject.cancelPreviousPerformRequests(withTarget: self)
306         UIApplication.shared.isNetworkActivityIndicatorVisible = true
307         self.perform(#selector(self.filterContent(for:)),
308                      with: searchText,
309                      afterDelay: 0.5)
310     }
311
312     func searchBar(_ searchBar: UISearchBar, selectedScopeButtonIndexDidChange selectedScope: Int) {
313         searchType = SearchType(rawValue: selectedScope)!
314         tableView.reloadData()
315         filterContent(for: searchBar.text!)
316     }
317
318     func searchBarCancelButtonClicked(_ searchBar: UISearchBar) {
319         if searchType == .globalUsers && searchBar.text == "" || filteredUsers.isEmpty {
320             searchType = .contacts
321             searchBar.selectedScopeButtonIndex = searchType.rawValue
322             filterContent(for: "")
323         }
324     }
325
326 }
```

Рисунок 4.2 - Клас SearchUsersViewController

Пошук в проекті з реалізованою Новою архітектурою кількість коду становить майже у 4 рази менше, ніж у проекті з реалізованою архітектурою MVC.

4.3 Порівняння можливості впровадження нової функціональності

Із попередніх двох пунктів логічно проаналізувати тепер взагалі можливість розширення існуючих проектів та можливість впровадження нових функціональностей. Як видно з попередніх результатів порівнянь для двої проектів при впровадженні нових екранів було затрачено абсолютно різні кількості часу і написано різну кількість коду. Із цього слідує, що впровадження нової функціональності у проекті з реалізованою архітектурою

MVC викликає певні труднощі і тому це займає досить багато часу, бо по-перше, написання з 0 нового екрану займає багато часу, а по-друге, у проекті з самого початку була не продумана схема для можливого розширення застосунку, тому весь попередньо затрачений час не був перевикористаний у вигляді базових класів і т.п.

Також в ситуації, якщо потрібно змінити вже існуючу логіку на щось нове, то в проекті з MVC на це буде затрачено багато часу, бо треба буде пройтись по всім місцям, де використовується стара логіка і змінити вручну її на нову. В проекті з Новою Архітектурою потрібно буде просто замінити стару логіку на нову в одному базовому класі, так як цей базовий клас використовують всі конкретні реалізації екранів в застосунку.

4.3 Покриття Unit-тестам

Модульне тестування, іноді блочне тестування або юніт-тестування (англ. Unit testing) - процес в програмуванні, що дозволяє перевірити на коректність окремі модулі вихідного коду програми, набори з одного або більше програмних модулів разом з відповідними керуючими даними, процедурами використання і обробки. [13]

Ідея полягає в тому, щоб писати тести для кожної нетривіальною функції або методу. Це дозволяє досить швидко перевірити, чи не призвело чергову зміну коду до регресії, тобто до появи помилок в уже відтестованих місцях програми, а також полегшує виявлення і усунення таких помилок.

Мета модульного тестування - ізолювати окремі частини програми і показати, що окремо ці частини працездатні.

Цей тип тестування зазвичай виконується програмістами.

Заохочення змін - модульне тестування пізніше дозволяє програмістам проводити рефакторинг, будучи впевненими, що модуль як і раніше працює коректно (регресійні тестування). Це заохочує програмістів до змін коду, оскільки досить легко перевірити, що код працює і після змін.

Спрощення інтеграції - модульне тестування допомагає усунути сумніви з приводу окремих модулів і може бути використано для підходу до тестування «знизу вгору»: спочатку тестуючи окремі частини програми, а потім програму в цілому.

Документування коду - модульні тести можна розглядати як «живий документ» для тестованого класу. Клієнти, які не знають, як використовувати даний клас, можуть використовувати юніт-тест в якості прикладу.

Відділення інтерфейсу від реалізації - оскільки деякі класи можуть використовувати інші класи, тестування окремого класу часто поширюється на пов'язані з ним. Наприклад, клас користується базою даних; в ході написання тесту програміст виявляє, що тесту доводиться взаємодіяти з базою. Це помилка, оскільки тест не повинен виходити за кордон класу. В результаті розробник абстрагується від з'єднання з базою даних і реалізує цей інтерфейс, використовуючи свій власний mock-об'єкт. Це призводить до менш пов'язаному коду, мінімізуючи залежності в системі.

Дуже часто покриття тестами в застосунку є вимогою від бізнеса і навіть буває таке, що бізнес зазначає мінімальний можливий процент покриття тестами. В такій ситуації добре, якщо бізнес це зазначив ще перед самою розробкою застосунку, але якщо це було в казано вже на фінальній стадії розробки, то тут лиш залишається сподіватись, що обраний архітектурний підхід передбачає можливість покриття коду тестами.[14]

Отже, як було заначено раніше, одним із головних недоліків архітектури MVC є неможливість покриття тестами. Тому, при проектуванні Нової архітектури це було враховано.

Для порівняння можливості покриття тестами в двох проектах був затрачений час в кінці розробки застосунків на покриття тестами. Мета – покрити тестами якомога більше класів за обмежений час.

Результат покриття тестами в процентах в проекті з MVC відображений на рисунку 4.3. Отже, за обмежений час було спробувано покрити тестами 14

класів, із них 4 взагалі не вдалось покрити жодним тестом, 7 з них було частково покрито тестами і тільки 3 вдалось повністю покрити тестами.

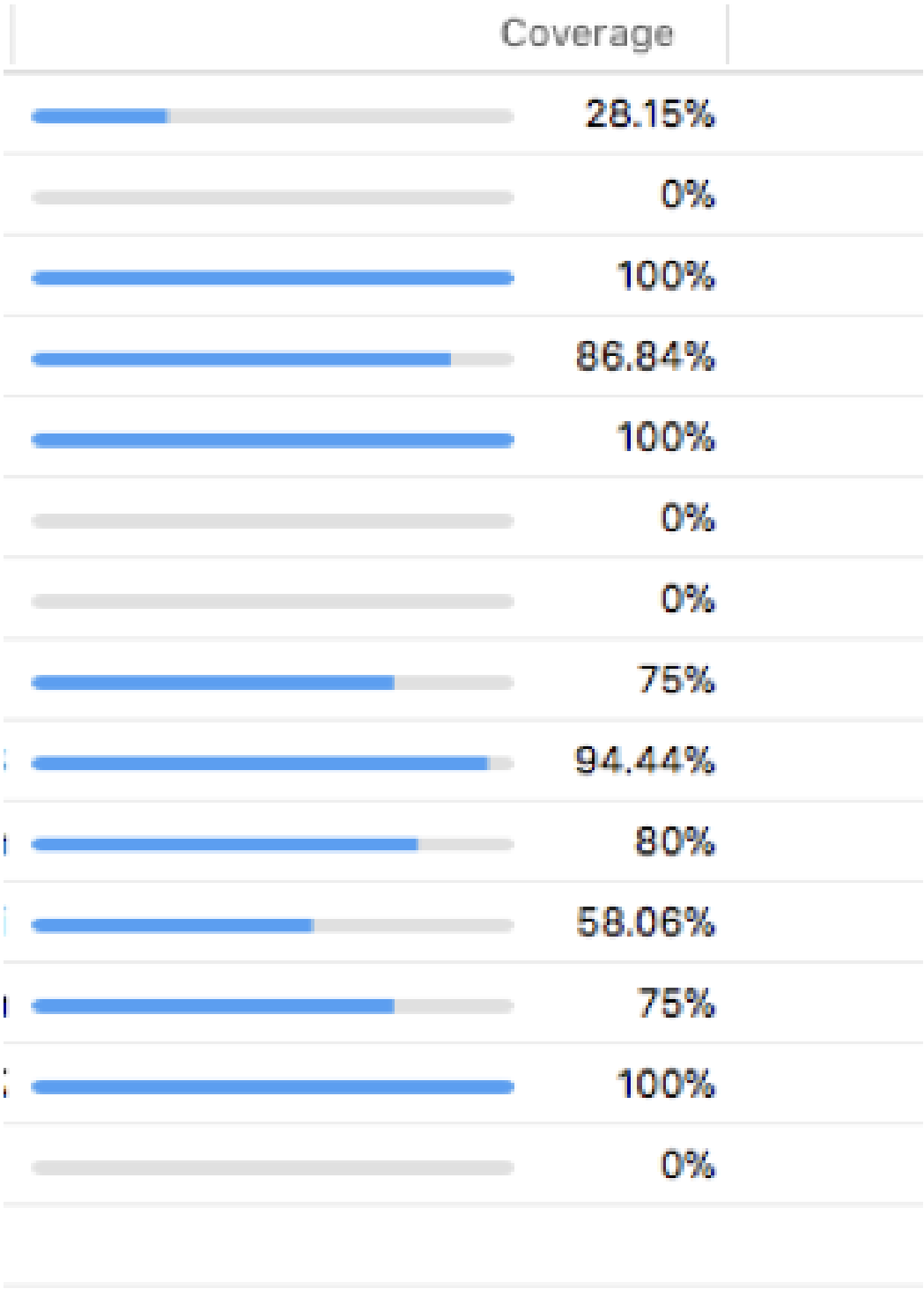


Рисунок 4.3 – Покриття тестами в проекті з MVC

Результат покриття тестами в процентах в проекті з Новою архітектурою

відображений на рисунку 4.4

Отже, за обмежений час було спробувано покрити тестами 17 класів і всі з них вдалось повністю покрити unit-тестами.

Рисунок 4.4 – Покриття тестами в проекті з Новою архітектурою

4.4 Висновки до розділу

Для доведення ефективності Нової Архітектури було запропоновано провести порівняння зі вже існуючою архітектурою Model-View-Controller, так як на сьогодні це найпопулярніший архітектурний підхід серед розробників.

Отже, було запропоновано і виконано порівняння двох архітектурних підходів Model-View-Controller і Нової Архітектури, яке складалось із чотирьох пунктів: затрачений час для розробки застосунку ZipHub, порівняння кількості коду в основних класах, порівняння можливості впровадження нової функціональності, покриття Unit-тестами. Після кожного етапу порівняння були отримані результати по кожному із проектів і в результаті проект з Новою Архітектурою показав кращі результати: всі чотири етапи Нова Архітектура перемагала.

ВИСНОВОК

У даній дисертації було спроектовано новий архітектурний підхід для мобільних застосунків на платформі iOS для поширеного типу додатків, які зазвичай розробляються серед українських розробників.

Були розглянуті існуючі архітектурні підходи та виокремлено найпопулярніші серед них. Було проаналізовано їх переваги та недоліки. На основі виділених переваг і недоліків були поставлені задачі та сформовано цілі для Нової Архітектури.

Запропонована Нова Архітектура була схематично розглянута та детально описана. Були детально розглянуті основні компоненти архітектурного підходу та запропоновані ефективні приклади реалізації та розглянуті можливі альтернативні підходи.

Для демонстрації практичного рішення було розроблено мобільний застосунок ZipHub. В процесі розробки були детально описані всі реалізовані компоненти в коді. Для кращого візуального розуміння були представлені зв’язок та залежності між застосованим архітектурним підходом та кінцевими екранами застосунку.

Для доведення ефективності та переваги Нової Архітектури було проведено порівняння із вже існуючим архітектурним підходом Model-View-Controller. Всі етапи порівняння показали позитивний результат у сторону Нової Архітектури.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1) Архітектура програмного забезпечення [Електронний ресурс] – Режим доступу до ресурсу: http://fitm.nusta.edu.ua/mediawiki/index.php?title=Архітектура_програмного_забезпечення.

2) Мобільнии застосунок [Електроннии ресурс] Режим доступу: https://uk.wikipedia.org/wiki/Мобільнии _додаток

3) What Is the iPhone OS (iOS)? [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://www.lifewire.com/what-is-ios-1994355.

4) Xcode [Електронний ресурс] – Режим доступу до ресурсу: https://developer.apple.com/xcode/.

5) Swift vs Objective-C Development Language: Which One is More Suitable in 2017? [Електронний ресурс]. – 2017. – Режим доступу до ресурсу: https://www.cleveroad.com/blog/swift-vs-objective-c--what-is-the-best-language-for-ios-development-.

6) Мартін Р. Чиста архітектура / Роберт Мартін., 2019. – 368 с.

7) MVC Framework - Introduction [Електронний ресурс] – Режим доступу до ресурсу: https://www.tutorialspoint.com/mvc_framework/mvc_framework_introduction.htm.

8) Potel M. MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java [Електронний ресурс] / Mike Potel. – 1996. – Режим доступу до ресурсу: http://www.wildcrest.com/Potel/Portfolio/mvp.pdf.

9) Gossman J. Tales from the Smart Client: Advantages and disadvantages of M-V-VM [Електронний ресурс] / John Gossman. – 2006. – Режим доступу до ресурсу: https://blogs.msdn.microsoft.com/johngossman/2006/03/04/advantages-and-disadvantages-of-m-v-vm/.

10) Core Data [Електронний ресурс] – Режим доступу до ресурсу: https://developer.apple.com/documentation/coredata.

11) Stevenson S. Build a Core Data App [Електронний ресурс] / Scott Stevenson – Режим доступу до ресурсу: http://cocoadevcentral.com/articles/000085.php.

12) Create a Table View [Електронний ресурс] – Режим доступу до ресурсу: https://developer.apple.com/library/archive/referencelibrary/GettingStarted/DevelopiOSAppsSwift/CreateATableView.html.

13) Модульное тестирование. Зачем, как и кто [Електронний ресурс] – Режим доступу до ресурсу: https://software-testing.ru/library/testing/general-testing/77-2008-09-29-07-30-13.

14) Толмачев Д. Проблема дублирования и устаревания знания в mock-объектах или Интеграционные тесты — это хорошо [Електронний ресурс] / Дмитро Толмачев. – 2016. – Режим доступу до ресурсу: https://habr.com/en/post/275249/.

ДОДАТОК А

ГРАФІЧНИЙ МАТЕРІАЛ